TURING 图灵程序设计丛书

Packt›

Functional Python Programming, Second Edition

Python函数式编程（第2版）

[美] 史蒂文·洛特 著　李超 陈文浩 译

■ 借鉴函数式编程的优点，开发简洁易读的高性能Python程序


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

史蒂文・洛特
（Steven F. Lott）

软件工程师、架构师、技术作家，20世纪70年代开始编程生涯，参与过100多个各种规模的项目研发。在使用Python解决业务问题方面，也有十余年经验。另著有《Python面向对象编程指南》等。

李超

工学博士，2010年毕业于北京理工大学。主要研究数据分析在工业信息化领域的应用以及函数式编程理论和方法。从Turbo Pascal on DOS到Python、R on Linux，沉迷编程二十余载，其间点滴心得，汇集在博客leetschau.github.io中。热衷于推广编程和开源数据分析技术，不定期组织线下交流活动。

陈文浩

计算机科学博士，1Token量化投研技术总监，具有多年专业投资研究经验，长期主导研发基于C++和Python的高性能量化交易系统，具备丰富的调试优化技巧，对底层框架较为熟悉。

数字版权声明



图灵程序设计丛书

Python函数式编程

（第2版）

[美] 史蒂文 · 洛特 著　李超 陈文浩 译

Functional Python Programming
Second Edition

人 民 邮 电 出 版 社
北　京

图书在版编目（CIP）数据

Python函数式编程 : 第2版 / （美）史蒂文·洛特（Steven F. Lott）著 ; 李超，陈文浩译. -- 北京 : 人民邮电出版社，2019.10
（图灵程序设计丛书）
ISBN 978-7-115-52017-3

Ⅰ. ①P… Ⅱ. ①史… ②李… ③陈… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2019)第197226号

内 容 提 要

Python 具备函数式编程的许多核心特征，因此可以借鉴其他函数式语言的设计模式和编程技术，编写出简洁优雅的代码。本书首先介绍函数式编程的一般概念及特点，然后讲解迭代器、生成器表达式、内置函数、常用高阶函数、递归与归约、实用模块和装饰器的用法，以及避开 Python 严格求值顺序的变通方法、Web 服务设计方法和一些优化技巧。

本书适合 Python 开发人员阅读。

◆ 著　[美] 史蒂文·洛特
译　李 超　陈文浩
责任编辑　杨 琳
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：18
字数：425千字　2019年 10 月第 1 版
印数：1－3 000册　2019年 10 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-8910号

定价：79.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

前　言

函数式编程为创建代码简洁明了的软件提供了许多技术。虽然 Python 不是纯粹的函数式语言，但仍然可以使用 Python 进行函数式编程。

Python 具备函数式编程的许多核心特征，使得我们可以借鉴其他函数式语言的设计模式和编程技术，编写出简洁优雅的代码。尤其值得一提的是 Python 的生成器表达式，使用它可以避免在内存中创建大型数据结构，通过降低资源消耗来提高执行速度。

Python 缺少创建纯粹函数式程序所需的一些语言特征，例如无限递归、针对所有表达式的惰性求值（lazy evaluation）以及优化编译器等。

函数式编程的许多核心要素都在 Python 中有所体现，例如函数是头等对象。Python 还提供了许多典型的高阶函数，例如广泛使用的内置 `map()`、`filter()` 和 `functools.reduce()` 等，以及不那么明显的 `sorted()`、`min()` 和 `max()` 等。

本书通过 Python 语言诠释函数式编程的核心思想，旨在利用函数式编程的优点，编写出代码简洁明了的 Python 程序。

目标读者

如果你希望借鉴函数式编程语言的技术和设计模式编写出简洁明了的 Python 程序，本书便是为你准备的。某些算法用函数式方法编写更为简洁，我们可以也应该运用函数式方法编写出更易读且更易维护的 Python 程序。

可以通过函数式编程在适宜的场景中开发出高性能的算法，但 Python 往往会生成大型中间数据结构，耗尽机器的内存和 CPU。因此常用生成器表达式代替大型列表，前者在保证可读性的前提下，内存消耗更少，运算速度更快。

本书内容

第 1 章：函数式编程概述，介绍 Python 中函数式编程对应的技术和语言特征，以及函数式设计为 Python 程序带来的好处。

第 2 章：函数式编程的特点，分析函数式编程范式的 6 个核心特征，以及每个特征在 Python 中的实现方法，还会讲到一些在 Python 中不易实现的函数式语言特征，例如为了支持编译优化，许多语言的类型匹配规则非常复杂。

第 3 章：函数、迭代器和生成器，介绍如何在 Python 中使用不可变对象和生成器表达式，如何将函数式编程的核心思想应用于 Python 和 Python 内置的集合类型，以及如何将函数式编程理念运用于这些数据结构。

第 4 章：使用集合，介绍如何使用 Python 的内置函数操作数据集。其中重点介绍几个比较简单的函数，例如 `any()` 和 `all()`，它们的共同点是能将集合转换为单个值。

第 5 章：高阶函数，介绍一些常用的高阶函数，例如 `map()` 和 `filter()`，以及如何创建新的高阶函数。

第 6 章：递归与归约，介绍如何使用递归设计算法，以及使用 `for` 循环提升性能，还会介绍其他一些应用广泛的归约函数，例如 `collections.Counter()`。

第 7 章：元组处理技术，介绍使用不可变的元组和命名元组代替状态可变对象的方法。相比而言，不可变对象没有误用属性导致对象行为异常（不连续或无效）的问题，用起来更简单可靠。

第 8 章：itertools 模块，介绍 Python 标准库中处理集合和生成器的几个函数，可用于简化处理集合数据的程序。

第 9 章：高级 itertools 技术，介绍 `itertools` 模块中不太常用的组合器函数，并演示错误使用这些函数导致的组合器膨胀问题。

第 10 章：functools 模块，介绍如何将 `functools` 模块中的函数用于函数式编程。此模块中适用于构建装饰器的少数函数留待第 11 章讨论。本章还会介绍一些支持其他函数式编程的函数。

第 11 章：装饰器设计技术，介绍如何用装饰器构建复合函数。虽然使用装饰器能给程序开发带来很大的灵活性，但也有概念限制：过于复杂的装饰器非但无用，还会严重降低程序的可读性。

第 12 章：multiprocessing 和 threading 模块，介绍函数式编程的一大优势：便于分流任务负载。使用不可变对象能避免设计欠佳的同步写入操作导致运行结果不可预料。

第 13 章：条件表达式和 operator 模块，介绍避开 Python 严格求值顺序的一些变通方法及其局限性，以及使用 `operator` 模块给某些简单的处理带来的轻微提升。

第 14 章：PyMonad 库，介绍 PyMonad 库的主要特点以及更丰富的函数式编程手段，还有单子（monad）。在一些函数式语言中，代码优化会打乱某些操作的顺序，而开发者可以使用单子强制程序按照期望的顺序执行。由于 Python 按照严格的顺序对表达式和声明求值，因此在

Python 中，单子的理论研究价值高于实用价值。

第 15 章：Web 服务的函数式设计方法。如果把 Web 服务看作从请求到响应的转换，那么可以把开发 Web 服务看作开发能实现该转换的一组函数。本章将介绍如何借助函数式编程方法构建响应式动态 Web 内容。

第 16 章：优化与改进，介绍提升程序性能的一些方法和技巧。在适合的场景中，这些方法（例如内存化）不但易于实现，并且能显著提升程序性能。

如何使用本书

阅读本书需要读者对 Python 3 和应用开发有基本了解。本书不会涉及 Python 中细微、复杂的语言特性，也不需要读者了解实现语言功能的内部机制。

读者需要对函数式编程有基本了解。由于 Python 不属于函数式语言，因此本书不会深入探讨函数式编程的概念，而会着重介绍其中适用于 Python 并有实用价值的部分。

书中的部分示例使用**探索性数据分析**（EDA）引出问题，演示函数式编程的特点。对统计学和概率论有基本了解有助于理解问题。只有很少一部分示例涉及数据科学。

你的计算机上需要安装并运行 Python 3.6。关于 Python 的更多信息，请访问 http://www.python.org。本书的示例代码经常使用类型提示，所以请安装最新版本的 mypy。关于最新版本的 mypy，请访问 https://pypi.python.org/pypi/mypy。

第 9 章的示例代码使用了 PIL 和 BeautifulSoup 4。为了保持版本兼容，使用了 PIL 库的新分支版本 Pillow 代替原始 PIL 库，详情请访问 https://pypi.python.org/pypi/Pillow/2.7.0 和 https://pypi.python.org/pypi/beautifulsoup4/4.6.0。

第 14 章的示例代码使用了 PyMonad 库，详情请访问 https://pypi.python.org/pypi/PyMonad/1.3。

可以通过如下命令安装以上所有库。

```
$ pip install pillow beautifulsoup4 PyMonad
```

下载源代码

如果你是从 http://www.packtpub.com 网站购买的图书，登录自己的账号后就可以下载所有已购图书的示例代码。如果你是从其他地方购买的图书，请访问 http://www.packtpub.com/support 网站并注册，我们会将代码文件直接发送到你的电子邮箱。

你也可以通过以下步骤下载代码文件。

(1) 在我们的网址上登录或注册。

(2) 选择 SUPPORT 标签。
(3) 点击 Code Downloads & Errata。
(4) 在 Search 框中输入书名并按屏幕上的提示操作。

文件下载后，使用以下工具的最新版本来解压缩或提取文件夹。

- WinRAR / 7-Zip（Windows）
- Zipeg / iZip/UnRarX（Mac）
- 7-Zip / PeaZip（Linux)

本书代码也托管在 GitHub 上，访问 https://github.com/PacktPublishing/Functional-Python-Programming-Second-Edition/即可获取[①]。Packt 拥有丰富的图书和视频资源，相关代码见 GitHub 仓库：https://github.com/PacktPublishing/。欢迎查阅！

排版约定

本书如下约定文本样式。

正文中的代码采用以下样式：“Python 有其他声明，如 `global` 或 `nonlocal`，用于在特定命名空间中修改变量的规则。”

代码块的样式如下所示。

```
s = 0
for n in range(1, 10):
    if n % 3 == 0 or n % 5 == 0:
        s +=n
print(s)
```

如果代码块的特定部分需要注意，相应的行或项会加粗。

```
s = 0
for n in range(1, 10):
    if n % 3 == 0 or n % 5 == 0:
        s += n
print(s)
```

命令行或输出如下所示。

```
$ pip install pillow beautifulsoup4 PyMonad
```

新的术语和**重要的词语**将显示为黑体。在屏幕上（如菜单或对话框中）出现的文字按如下样式显示：“在现有的许多范式中，我们重点区分**函数式编程**和**命令式编程**。”

① 你可以直接访问本书中文版页面，下载本书项目的源代码：http://www.ituring.com.cn/book/2658。——编者注



此图标表示警告或需要特别注意的内容。



此图标表示提示或技巧。

问题与反馈

一般反馈：发送邮件至 feedback@packtpub.com 并在主题处注明书名。如果对于本书有任何方面的疑问，请发送邮件至 questions@packtpub.com。

勘误：尽管我们尽力确保内容准确，但出错仍在所难免。如果你在书中发现错误，不管是文本还是代码，希望能告知我们，我们不胜感激。[①]如果你发现任何错误，请访问 http://www.packtpub.com/submit-errata, 选择书名，点击 Errata Submission Form 链接，并输入详细说明。

反盗版：如果你发现我们的作品在互联网上被以任何形式非法复制，请立即为我们提供地址或网站名称，非常感谢。请把可疑盗版材料的链接发到 copyright@packtpub.com。

成为作者：如果你掌握某个领域的专业知识，并且有兴趣写作图书，请访问 authors.packtpub.com。

评论

请给出评论。阅读、使用本书后，请在购买网站上留下评论。这样潜在读者可以参考你的意见来决定是否购买，Packt 可以了解到你对该产品的看法，作者也能看到你对本书的反馈。谢谢！

想了解关于 Packt 的更多信息，请访问 packtpub.com。

电子书

扫描如下二维码，即可购买本书电子版。

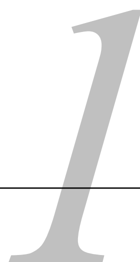


① 针对本书中文版的勘误，请到 http://www.ituring.com.cn/book/2658 查看和提交。——编者注

目　　录

第 1 章

函数式编程概述



函数式编程通过在函数中定义表达式和对表达式求值完成计算。它尽量避免由于状态变化和使用可变对象引入复杂性，让程序变得简洁明了。本章将介绍函数式编程的一些基本技术，以及如何在 Python 中运用这些技术。最后会介绍通过这些设计模式构建 Python 应用时，函数式编程带来的好处。

Python 包含大量函数式编程特征，但它不是纯粹的函数式编程语言。它不仅具备函数式编程的诸多优势，还保留了命令式编程的强大优化能力。

本书中的许多示例来自 EDA 领域。使用函数式编程方式解决该领域的问题可以很好地展示其特点，而且与其他解决方法相比有明显优势。

本章的主要目标是介绍函数式编程的基本原则，第 2 章开始编写 Python 代码。



本书主要使用 Python 3.6 作为实现语言，部分示例也可以在 Python 2 中运行。

1.1 编程范式

编程范式并没有统一的划分标准。本书重点分析其中两个范式：**函数式编程**和**命令式编程**。二者最重要的特征区别是**状态**。

在命令式语言（比如 Python）中，计算的状态是通过不同命名空间中变量的值反映的。变量的值决定计算的当前状态，一条语句通过增加或改变（甚至是删除）变量来改变当前状态。“命令式”语言的每一条语句都是一个通过某种方式改变状态的命令。

本书主要关注赋值语句以及它如何改变状态。除此之外，Python 中的其他语句包括用于在命令空间中改变变量规则的 `global`、`nonlocal` 等，用于改变变量所处语境的 `def`、`class` 和 `import` 等，用作判断条件确定一组语句如何改变计算状态的 `try`、`except`、`if`、`elif` 和 `else` 等。而循环语句，如 `for` 和 `while`，则是将一组语句作为整体以重复改变计算状态。可见所有

这些语句都是通过某种方式改变变量状态的。

理想状态下，每一条语句通过改变状态，推动计算从初始状态向期望的最终结果不断靠近。然而，这种“推动计算一步步向前”的模式难以验证。需要首先定义出最终状态，找到能达到该状态的语句，从而推导出达到该状态需要的前提条件，然后重复上述步骤，直到找到一个可接受的初始状态。

在函数式语言中，使用“对函数求值”这一更简单的概念代替改变变量值的“状态”，每次对函数求值都会在现有对象的基础上创建一个或多个新对象。函数式程序即函数的组合，相应的开发过程是：首先设计一组易于理解的底层函数，然后在此基础上设计符合业务需求的高级函数。相比于由复杂的流程控制组成的指令集合，高级函数更容易可视化。

形式上，函数求值更接近算法的数学表达。以简单的代数形式设计算法，便于处理特殊情况和边界条件，而且函数更有可能按照预期工作，也便于编写单元测试用例。

请注意，通常函数式程序比功能相同的命令式（面向对象或者过程式的）程序更加简洁明了和高效，但这些优点并不是自然而然的，需要仔细地设计，但付出的努力通常少于设计功能类似的过程式程序。

1.2　细分过程范式

命令式编程可以再细分为多个子类别，本节简单介绍过程式编程和面向对象编程的区别，并重点讲解面向对象属于命令式编程的原因。过程式编程和面向对象编程虽然有区别，但它们与函数式编程的差别更大。

下面通过代码示例解释这些概念。有些人觉得这么做是在重新造轮子，然而这其实是抽象概念的具体表现。

对于某些计算过程，完全可以忽略 Python 的面向对象特点，只使用简单的数值计算。例如用下面的方法可以得到一组属性相同的数。

```
s = 0
for n in range(1, 10):
    if n % 3 == 0 or n % 5 == 0:
        s += n
print(s)
```

和 `m` 仅包括 3 或 5 的倍数。以上程序是严格过程式的，避免使用 Python 的任何面向对象特征。程序的状态由变量 `s` 和 `n` 定义，变量 `n` 的取值范围是 1~10，每次循环中 `n` 的值依次增加，可以确定 `n == 10` 时循环结束。使用类似的原始数据结构，完全可以用 C 或者 Java 编写出功能相同的程序。

下面利用 Python 的**面向对象编程**（object-oriented programming，OOP）特征编写一段类似的代码。

```
m = list()
for n in range(1, 10):
    if n % 3 == 0 or n % 5 == 0:
        m.append(n)
print(sum(m))
```

程序运行结果与前面的结果相同，但它内部维护了一个状态可变的集合对象 `m`，计算状态由 `m` 和 `n` 定义。

`m.append(n)`和 `sum(m)`令人费解的语法让一些开发者误以为 Python 不是纯粹的面向对象语言：它混合了 `function()`和 `object.method()`两种语法。然而事实上 Python 是纯粹的面向对象语言，一些语言（例如 C++）允许使用非对象的原始数据类型，例如 `int`、`float` 和 `long`。Python 中没有原始数据类型，前缀的语法形式不会改变语言的本质。

严格地说，完全可以采用纯粹的面向对象风格，基于 `list` 类生成一个包含 `sum` 方法的子类。

```
class Summable_List(list):
    def sum(self):
        s = 0
        for v in self:
            s += v
        return s
```

接下来使用 `Summable_List()`类代替 `list()`方法初始化变量 `m`，就可以用 `m.sum()`方法代替 `sum(m)`方法来对 `m` 求和了。该示例可以证明 Python 是纯粹的面向对象语言，前缀的使用仅是语法糖而已。

前面 3 个例子都基于变量值显式确定程序的状态，使用赋值语句改变变量值，推动计算前进。我们可以在程序中插入 `assert` 语句，确保程序状态完全按照要求变化。

关键之处不是命令式编程存在某种缺陷，而是函数式编程是一种思维方式的转变，这种改变适用于许多场景。下面介绍如何用函数式方法编写同一个算法，你会发现函数式编程并没有使算法显著变短或变快。

1.2.1　使用函数式范式

在函数式编程中，求 3 或 5 的倍数可分为两部分。

- 对一系列数值求和。
- 生成一个满足某个条件的序列，例如 3 或 5 的倍数组成的序列。

一个列表的和的递归形式定义如下。

```
def sumr(seq):
    if len(seq) == 0: return 0
    return seq[0] + sumr(seq[1:])
```

可以把序列的和分为两种情况。**基础形式**：一个长度为 0 的序列，和为 0。**递归形式**：序列的和等于序列中的第一个元素加上序列中后续元素的和。

由于递归形式的序列长度小于原序列，所以任何长度有限的序列最终都会退化为基础形式。

该函数运行示例如下。

```
>>> sumr([7, 11])
18
>>> 7+sumr([11])
18
>>> 18+sumr([])
0
```

第一个例子计算了包含多个值的列表之和。第二个例子演示了递归规则将第一个值 `seq[0]` 和后续所有值的和 `seq[1:]` 相加。最后一个计算包含了对空列表求和，其值定义为 0。

这个例子中，代码最后一行的+运算符和初始值 0 表明其为求和。如果将运算符从+改为*，将初始值从 0 改为 1，则表明其为序列乘积。后面会详细介绍这种抽象化方法。

对于一列值，可以用类似的方法递归，定义如下。

```
def until(n, filter_func, v):
    if v == n: return []
    if filter_func(v): return [v] + until(n, filter_func, v+1)
    else: return until(n, filter_func, v+1)
```

该函数的基础形式为：给定一个值 `v` 和一个上限 `n`，如果 `v` 达到上限，则返回一个空列表。

根据 `filter_func()` 函数的不同返回值，递归形式有两种情况。如果 `v` 通过了 `filter_func()` 函数的测试，返回一个序列，则该序列的第一个元素是 `v`，后续元素由 `until()` 作用于后续序列的返回值组成。如果 `v` 没有通过 `filter_func()` 函数的测试，将忽略该值，返回值由函数作用于剩余元素得到的值组成。

可以看到 `v` 在每次递归中递增，直到达到上限 `n`，也就是基础形式。

下面介绍如何使用 `until()` 函数生成 3 或 5 的倍数。首先定义一个用于筛选数值的 `lambda` 对象。

```
mult_3_5 = lambda x: x % 3 == 0 or x % 5 == 0
```

（这里使用 `lambda` 定义简单函数是为了保持简洁。如果函数比较复杂，多于一行代码，请使用 `def` 语句。）

从命令提示符界面观察 lambda 的行为，如下所示。

```
>>> mult_3_5(3)
True
>>> mult_3_5(4)
False
>>> mult_3_5(5)
True
```

结合 until()函数，它可以生成一系列 3 或 5 的倍数。

使用 until()函数生成一系列值，如下所示。

```
>>> until(10, lambda x: x % 3 == 0 or x % 5 == 0, 0)
[0, 3, 5, 6, 9]
```

然后可以使用之前定义的递归版 sum()函数计算一系列数值的和了。这里将用到的所有函数，包括 sum()、until()和 mult_3_5()，都定义为递归的，计算时不需要使用临时变量保存计算状态。

之后还会多次用到这种纯粹递归风格的函数来定义思想。请注意，许多函数式语言的编译器可以优化此类简单的递归函数，但 Python 不会进行此类优化。

1.2.2　使用混合范式

下面介绍如何用函数式编码实现前面计算 3 或 5 的倍数的例子。**混合型**函数的实现代码如下所示。

```
print(sum(n for n in range(1, 10) if n % 3 == 0 or n % 5 == 0))
```

这里使用了**嵌入式生成器表达式**迭代数值集合，并计算它们的和。range(1, 10)方法是可迭代的，所以它是一种生成器表达式，返回一个数值序列 $\{n|1 \leqslant n < 10\}$。n for n in range(1, 10) if n % 3 == 0 or n % 5 == 0 稍复杂一些，但它也是可迭代表达式，返回一个数值集合 $\{n|1 \leqslant n < 10 \wedge (n \bmod 3 = 0 \vee n \bmod 5 = 0)\}$。变量 n 与集合中的每个值绑定，表示集合的内容，而非当前的计算状态。sum()方法的输入值是一个可迭代表达式，输出最终计算结果：对象 23。



绑定变量仅存在于生成器表达式内部，上述程序中，变量 n 在生成器表达式之外是不可见的。

可以将表达式中的 if 从句看作独立的函数，便于用其他函数替换它。可以使用一个名为 filter()的高阶函数代替上面生成器表达式中的 if 从句，第 5 章会详细介绍高阶函数。

这个例子中的变量 n 不同于前面两个命令式实现中的变量 n，生成器表达式之外的 for 语句在本地命名空间中创建变量，而生成器表达式并不创建 for 语句式的变量。

```
>>> sum(n for n in range(1, 10) if n % 3 == 0 or n % 5 == 0)
23
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

生成器表达式绑定的范围外不存在变量 n，即它并不定义计算状态。

1.2.3　对象的创建过程

在某些情况下，观察中间对象有助于理解计算过程，但请注意，计算的路径并不是唯一的。当函数满足交换律和结合律的时候，改变求值顺序会创建出不同的中间对象。通过这种方式，可以在保证计算正确性的同时提升计算性能。

以下面这个表达式为例：

```
>>> 1+2+3+4
10
```

下面讲解不同的计算过程是如何得到相同的计算结果的。由于+运算符满足交换律和结合律，有许多条计算路径都能得到相同结果。

根据选择待计算值顺序的不同，有以下两种主要的计算路径。

```
>>> ((1+2)+3)+4
10
>>> 1+(2+(3+4))
10
```

第一种情形是从左向右结合并求值，此时对象 3 和 6 作为求值过程的中间对象被创建出来。

第二种情形则是从右向左结合并求值，中间对象是 7 和 9。在这个简单的整数算术运算中，两种方式的表现相同，优化无助于提升性能。

涉及数组的追加操作时，改变结合方式可能会提升性能。

示例如下。

```
>>> import timeit
>>> timeit.timeit("((([]+[1])+[2])+[3])+[4]")
0.8846941249794327
>>> timeit.timeit("[]+([1]+([2]+([3]+[4])))")
1.0207440659869462
```

可以看到，从左向右计算性能更佳。

对于函数式编程的设计，以任意顺序使用+运算符（或者 add()函数），结果不变，即+运算符不影响使用方式。

1.2.4　乌龟塔

严格意义上，Python 的函数式编程并非函数式的，Python 不是 Haskell、OCaml 或 Erlang。请注意，真正完成计算过程的处理器硬件本身就不是函数式的，甚至严格意义上不是面向对象的，CPU 实际上是过程式的。

> 所有的编程语言都基于抽象、库、框架和虚拟机，这里的抽象又基于更底层的抽象、库、框架和虚拟机。有个很形象的比喻：整个世界被一只大乌龟驮在背上，这只大乌龟又被另外一只更大的乌龟驮在背上，这只更大的乌龟又被一只比它还大的乌龟驮在背上……
>
> 一言以蔽之：全是乌龟。
>
> ——佚名

抽象形成的层是无尽的。

更重要的是，这种抽象和虚拟机并不会影响通过 Python 的函数式特性设计软件。

即使在函数式语言内部，也存在更纯粹的语言和不太纯粹的语言。有些语言经常使用 `monads` 处理像文件系统输入、输出这样有状态的事务，另外一些语言则使用类似于 Python 的混合型环境，通过仔细地隔离含有状态的过程式动作来设计函数式的软件。

本书的函数式 Python 编程基于以下 3 层抽象。

- 应用由函数组成，直到层层分解碰到对象。
- 支撑函数式编程的 Python 运行时环境是由对象组成的，直到层层分解碰到库。
- 支撑 Python 运行的库就是驮着 Python 的乌龟。

更底层的操作系统和硬件有它们各自的乌龟塔，而且与我们要处理的问题无关。

1.3　函数式编程经典示例

本节基于 John Hughes 的论文“Why Functional Programming Matters”，来分析一个函数式编程的经典实例，这篇文章出自论文集 *Research Topics in Functional Programming*。

此论文深入分析了函数式编程，并提供了几个例子，我们只分析其中的一个：用 Newton-Raphson 算法求解函数（平方根函数）。

该算法的许多实现都是通过 `loops` 显式管理状态的，比如 Hughes 的论文中就给出了一段 Fortran 代码，通过有状态的命令式流程求解。

算法的主体是如何根据当前的近似值计算出下一个近似值。函数 `next_()` 以 `sqrt(n)` 的当前

近似值 x 为参数，计算出下一个近似值，并确保最终解就在之前近似值的范围内，代码如下所示。

```
def next_(n, x):
    return (x + n / x) / 2
```

该函数计算出一系列值 $a_{i+1} = \frac{(a_i + n / a_i)}{2}$，相近两个值的距离每次迭代减半，所以会迅速收敛到 $a = \frac{n}{a}$，即 $a = \sqrt{n}$。这里没有将迭代函数命名为 next()，以避免与 Python 的内置函数发生冲突，使用 next_() 保证在不冲突的前提下尽量清晰地表达出函数的功能。

在命令提示符界面使用该函数，如下所示。

```
>>> n = 2
>>> f = lambda x: next_(n, x)
>>> a0 = 1.0
>>> [round(x,4) for x in (a0, f(a0), f(f(a0)), f(f(f(a0))),)]
[1.0, 1.5, 1.4167, 1.4142]
```

首先定义收敛到 $\sqrt{2}$ 的 lambda 表达式并赋值给变量 f，将变量 a_0 作为初始值，然后对一系列递归值求值：$a_1 = f(a_0)$、$a_2 = f(f(a_0))$，等等。将这些函数放在一个生成器表达式中，便于对返回值做指定精度的四舍五入，从而使计算结果更易读，并便于 doctest 使用。该序列会快速地向 $\sqrt{2}$ 收敛。

我们可以编写一个函数，生成一个含 a_i 的无限长序列，向平方根收敛。

```
def repeat(f, a):
    yield a
    for v in repeat(f, f(a)):
        yield v
```

该函数利用近似函数 f() 和初始值 a 生成近似值。如果把近似函数替换成前面定义的 next_() 函数，就可以得到关于参数 n 平方根的一系列近似值。



> 其中 repeat() 函数要求 f() 函数只有一个参数，而定义的 next_() 函数有两个参数。可以用一个匿名函数对象 lambda x: next_(n, x) 绑定其中一个变量，创建 next_() 函数的部分绑定版本。
>
> Python 的生成器函数不能自动实现递归，必须显式迭代递归结果，并一个一个单独生成计算结果。使用 return repeat(f, f(a)) 并不能多次循环生成一系列值，而会结束迭代并返回一个生成器表达式。

有两种方法可以返回一系列值，而不是生成器表达式。

- 编写显式 for 循环：for x in some_iter: yield x
- 使用 yield from 语句：yield from some_iter

从递归生成器表达式中返回结果，这两种方法的效果相同，这里倾向于使用 yield from 语句。不过在有些情况下，yield 结合复杂表达式，往往比相应的映射和生成器表达式更清晰。

当然，我们并不想计算无限长序列，只要两次迭代的近似值足够接近，就可以任取其中一个作为最终解。通常用希腊字母 ε 表示两个值足够接近，这里的含义是计算平方根的误差上限。

在 Python 中，我们需要设法从无限序列中一次取一个值，通常把复杂的递归包裹在简单的接口函数中，见如下代码片段。

```
def within(ε, iterable):
    def head_tail(ε, a, iterable):
        b = next(iterable)
        if abs(a-b) <= ε: return b
            return head_tail(ε, b, iterable)
    return head_tail(ε, next(iterable), iterable)
```

首先定义了内部函数 head_tail()，以误差允许范围 ε、可迭代序列中的一个值 a 和可迭代序列的剩余部分 iterable 为参数，iterable 的下一个值与变量 b 绑定。如果 $|a-b|\leqslant\varepsilon$，两个值距离足够近，表明已找到平方根的解；否则以 b 为参数，递归调用函数 head_tail()，以获取下一次迭代的近似值。

函数 within()只需要用参数 iterable 的第一个值初始化内部的 head_tail()函数，后面由递归自动完成。

有些函数式语言允许将一个值放回可迭代序列，在 Python 中，这类似于用 unget()或者 previous()方法将一个值追加到迭代器中，然而 Python 的可迭代数据结构并没有提供这种高级功能。

结合上面 3 个函数 next_()、repeat()和 within()，即可创建求平方根函数。

```
def sqrt(a0, ε, n):
    return within(ε, repeat(lambda x: next_(n,x), a0))
```

repeat()函数基于 next_(n, x)函数生成一个（可能的）无限长序列，当两次迭代值之差小于 ε 时，within()即停止序列继续生成值。

使用这个 sqrt()函数需要提供一个初始值 a0 和误差值 ε，表达式 sqrt(1.0, .0001, 3) 表示从初始估计值 1.0 开始计算 $\sqrt{3}$，误差值为 0.0001。对于大多数应用，初始值可以选择 1.0，不过初始值与实际平方根越接近，函数收敛越快。

以上近似算法的最初版本是用 Miranda 语言编写的，可以看到 Miranda 和 Python 的实现之间有一些显著区别，主要是 Miranda 可以构建 cons，可以通过类似于 unget 的方式将值放回可迭代对象中。Miranda 和 Python 的这种对比说明了 Python 适用于实现多种函数式编程技术。

1.4 EDA

本书后续章节的函数式编程示例大多来自 EDA 领域，该领域包含很多处理复杂数据集的算法和技术，函数式编程往往能很好地连接起问题领域和解决方案。

虽然每个人有自己的行事风格，但处理 EDA 领域的问题通常可以划分成下面几个阶段。

- **准备数据**：主要是抽取和变换源应用中的数据。例如解析原始数据格式，对数据执行某种程度的清洗（比如移除不可用数据和异常数据等），这是函数式编程擅长的领域。
- **数据探测**：对数据进行初始画像，通常使用一些基本的统计函数来完成，这也是函数式编程擅长的领域。用专业术语讲，该阶段我们关注数据的单变量和双变量统计特征，实际上就是数据的描述性统计特征值，包括平均值、中位数、众数等。数据探测还可能涉及数据可视化，但本书不探讨这个主题，因为它不怎么采用函数式编程。如果你感兴趣，可以尝试一些工具包，例如 SciPy。访问如下网址，可获取有关 SciPy 工作原理和使用方法的更多信息。
  - https://www.packtpub.com/big-data-and-business-intelligence/learning-scipy-numerical-and-scientific-computing
  - https://www.packtpub.com/big-data-and-business-intelligence/learning-python-data-visualization
- **数据建模与机器学习**：主要解决如何从已有模型中提取新数据，但本书不涉及，因为从数学角度看有些模型十分复杂，讨论这些问题无助于理解函数式编程。
- **评估与比较**：当存在多个可用模型时，就需要针对当前数据评估哪个模型更适合。此过程主要涉及计算模型常用的一些描述型统计特征值，函数式设计技术能有所帮助。

EDA 的目标是创建模型为应用决策提供依据。很多情况下，一个模型可能就是一个简单的函数。使用函数式编程方式，便于将已有模型应用于新数据，生成业务人员可以理解的结果。

1.5 小结

本章主要介绍了编程范式，并比较了函数式编程和另外两种常用的命令式编程范式的区别。本书旨在向读者介绍 Python 的函数式编程特征。我们讲到了 Python 并非纯粹的函数式编程语言，Python 函数式编程的指导思想是将简洁明了的函数式编程与性能优化相结合，形成有 Python 特色的混合式编程方法。

下一章将详细介绍函数式编程的 5 种基本技术，这些技术构成了 Python 混合函数式编程的核心要素。

第 2 章

函数式编程的特点

Python 内置了函数式编程的大部分特性。编写函数式的 Python 代码要求我们尽量避免使用命令式（包括过程式和面向对象式）编程技术。

本章将介绍以下函数式编程技术。

- 头等函数和高阶函数，也称“纯函数”。
- 不可变数据结构。
- 严格求值与非严格求值，也称“积极求值”与“惰性求值”。
- 用递归代替显式循环语句。
- 函数类型系统。

回顾一下上一章提到的概念，首先，纯粹的函数式编程避免了由于使用变量赋值导致程序显式维护计算状态而带来的复杂性；其次，Python 不是纯粹的函数式语言。

本书不会给出函数式编程的确切概念。Python 不是纯粹的函数式语言，并且严格的定义并无帮助。我们将关注公认的重要函数式特性，不涉足有争议的模糊地带。

本章的示例代码将涉及 Python 3 的类型提示语法。类型提示有助于开发者阐述函数定义的核心目标，这里使用 mypy 工具分析类型提示。与提供单元测试和代码静态分析的 pylint 类似，mypy 也是构建高质量软件工具链的重要组成部分。

2.1　头等函数

总体而言，函数式编程简洁明了，因为函数可以用作其他函数的参数或者返回值，后续会给出很多这样的例子。

要做到这一点，函数必须是运行时环境中的头等对象。在 C 等语言中，函数不是运行时中的对象，然而在 Python 中，函数通常是通过 `def` 语句创建的对象，且其他函数可以使用。我们还可以通过创建可调用对象，或者将 `lambda` 表达式赋给变量来创建函数。

创建函数即创建一个带有属性的对象，如下所示。

```
>>> def example(a, b, **kw):
...     return a*b
...
>>> type(example)
<class 'function'>
>>> example.__code__.co_varnames
('a', 'b', 'kw')
>>> example.__code__.co_argcount
2
```

这里我们创建了一个对象：`example`，其为 `function` 类。此对象包含很多属性，与该函数对象关联的`__code__`对象也含有自己的属性。其具体实现细节不重要，重要的是 Python 中的函数是头等对象，我们完全可以像处理其他对象一样处理函数，比如上面的代码示例展示了函数对象的其中两个属性。

2.1.1　纯函数

为了提高程序可读性，使用的函数要尽量没有副作用，即所谓的“纯函数”。使用纯函数的好处包括可以通过改变求值顺序实现优化，而其最重要的优势在于概念简单、测试方便。

在 Python 中，编写纯函数式代码要求代码的作用域为本地，具体而言，就是避免使用 `global` 语句。`nonlocal` 语句的使用也可能对作用域产生副作用，也应留意，虽然副作用限制在一个嵌套函数里。实际上，达到这些要求并不难。可以把纯函数看作普通的 Python 编程实践。

并没有简单的方法能保证 Python 函数没有副作用，编码时不小心违反了纯函数规则也是常有之事。如果实在担心可能违反规则，可以写一个函数，使用 `dis` 模块扫描给定函数的`__code__.co_code` 属性，即编译后的代码，检查是否包含全局引用。它能对内部闭包和`__code__.co_freevars` 元组方法的使用给出提示。然而为了避免极少出现的情形而运用这类复杂的技术有些得不偿失，因此后续不会展开讨论。

Python 的 `lambda` 表达式是纯函数。虽然不太推崇，但确实可以通过 `lambda` 表达式创建纯函数。

将 `lambda` 表达式赋给变量以创建函数的示例如下。

```
>>> mersenne = lambda x: 2 ** x - 1
>>> mersenne(17)
131071
```

将 `lambda` 表达式赋给变量 `mersenne`，即可得到一个纯函数，实际上是一个包含单一参数 `x`，并返回单个值的可调用对象。因为 `lambda` 表达式中不能包含赋值语句，所以它总为纯函数，适用于函数式编程。

2.1.2 高阶函数

使用高阶函数可以使程序简洁明了。高阶函数以其他函数为参数，或者用函数作为返回值。我们可以使用高阶函数将简单的函数组合成复合函数。

以 Python 的 max()函数为例，我们可以提供一个函数作为其参数，来改变 max()函数的行为。

待处理的数据如下。

```
>>> year_cheese = [(2000, 29.87), (2001, 30.12), (2002, 30.6), (2003,
30.66),(2004, 31.33), (2005, 32.62), (2006, 32.73), (2007, 33.5),
(2008, 32.84), (2009, 33.02), (2010, 32.92)]
```

可以如下所示使用 max()函数。

```
>>> max(year_cheese)
(2010, 32.92)
```

其默认行为会比较列表中的每个元组，按元组下标为 0 的元素比较大小，返回最大的元素所在的元组。

由于 max()函数是高阶函数，因此可以添加一个函数作为其参数。这里用一个 lambda 表达式作为它的函数参数，如下所示。

```
>>> max(year_cheese, key=lambda yc: yc[1])
(2007, 33.5)
```

在这个例子中，max()函数用 lambda 表达式定义的函数作为比较依据，返回了下标为 1 的最大元素所在的元组。

Python 提供了许多高阶函数，后面（主要是第 5 章）会介绍 Python 提供的许多高阶函数以及编写高阶函数的方法。

2.2 不可变数据结构

函数式编程中不能使用变量跟踪计算的状态，所以我们需要研究如何使用不可变对象，比如可以使用元组和命名元组构建复杂的不可变数据结构。

不可变对象的概念在 Python 中并不陌生。程序使用不可变元组比使用可变对象的性能要好。在某些情况下，使用不可变对象时，我们需要重新考虑算法，以避免改变对象所带来的开销。

我们将（几乎）完全避免使用类定义，虽然在一门**面向对象编程**的语言里这么做似乎不合逻辑。通过阅读本书你会了解，函数式编程并不需要有状态的对象。可以定义可调用对象，并通过它们把互相关联的函数放在同一个命名空间内，这类对象可以在多个级别上进行配置。通过可调用对象创建缓存也很简单，而且能大幅提升性能。

下面介绍一个处理不可变对象的常用设计模式：wrapper()函数。由元组组成的列表是常见的数据结构，我们经常用以下两种方式处理它们。

- **使用高阶函数**：如前所述，为 max()提供一个 lambda 表达式——max(year_cheese, key=lambda yc: yc[1])。
- **使用“打包-处理-拆包”模式**：在函数式语境中，这种模式可以表述为 unwrap(process(wrap(structure)))。

例如，看看以下命令片断。

```
>>> max(map(lambda yc: (yc[1], yc), year_cheese))[1]
(2007, 33.5)
```

这个例子很好地展示了上面说的三部曲模式：打包数据结构，获取打包后数据结构的最大值，然后拆包。

首先是打包。map(lambda yc: (yc[1],yc), year_cheese)把列表中的每一项转换成一个二元组，其中用于比较的项后面跟着原始项，这里用于比较的项是 yc[1]。

接下来用 max()函数处理逻辑。因为之前已经把需要比较的项变成了二元组的第一个元素，所以使用 max()的默认方式就可以了，不再需要它的高阶函数能力。

最后用下标[1]提取最终结果，即拆包。这里是从 max()返回的结果中通过取第二个元素得到目标元组。

这类打包、拆包的操作在函数式编程中很常用，所以有些函数式语言为这类操作提供了专门的函数，例如 fst()和 snd()等，这样就可以使用前缀式语法，而不必使用[0]或[1]这样的下标了。我们可以实现这些函数，并将其应用于“打包－处理－拆包”模式中，如下所示。

```
>>> snd = lambda x: x[1]
>>> snd(max(map(lambda yc: (yc[1], yc), year_cheese)))
(2007, 33.5)
```

上例中，通过定义 snd()函数实现取元组第二个元素的功能，这样实现的“打包－处理－拆包”模式更易读。我们使用 map(lambda... , year_cheese)打包原始数据项，使用 max()进行处理，最后用 snd()函数从返回的元组中提取第二个元素。

第 13 章将介绍上述基于 lambda 表达式的 fst()函数和 snd()函数的替代解决方案。

2.3　严格求值与非严格求值

函数式编程高效，原因之一是将计算推迟到需要的时候进行。惰性（也称“非严格”）求值非常重要，Python 内置了对它的支持。

Python 中，逻辑运算符 and、or 和 if-then-else 都是非严格的。有时也称之为“短路”运算符，因为它们不需要计算全部参数就能得到最终结果。

以下命令片断展示了 and 运算符的惰性求值特性。

```
>>> 0 and print("right")
0
>>> True and print("right")
right
```

执行上面的代码时，如果 and 运算符左边的表达式值为 False，不会对右边的表达式求值；只有当左边的表达式值为 True 时，才会对右边的表达式求值。

除此之外，Python 使用严格求值规则。除了逻辑运算符，表达式都是严格地从左向右求值的。一组语句也是严格按顺序求值的，列表字面量和元组亦然。

当创建一个类时，它的各个方法是严格按顺序定义的。在类的定义中，所有方法在创建之后（默认）被放入一个字典，并不会保持之前的顺序。如果在一个类中创建两个名字相同的方法，那么由于严格的求值顺序，只会保留后面的方法，前面定义的方法会被覆盖掉。

Python 的生成器表达式和生成器函数是惰性的，在求值时，这些表达式不会马上计算出所有的可能结果。如果不把计算过程显式打印出来，很难看到惰性求值的结果。下面的例子演示了通过引入带有副作用的 range() 函数生成值的过程。

```
def numbers():
    for i in range(1024):
        print(f"= {i}")
        yield i
```

每生成一个值，该函数就将其打印出来，以此给出调试提示。如果这个函数是严格求值的，将会打印出所有 1024 个值，但由于它是惰性的，所以只会按需生成值。



Python 2 的 range() 函数是严格求值的，创建后就会生成所有包含的值。Python 3 的 range() 函数是惰性求值的，不会创建大型数据结构。

可以用惰性求值的方式使用这个带日志功能的 numbers() 函数。下面编写一个只求部分值（而非全部）的函数。

```
def sum_to(n: int) -> int:
    sum: int = 0
    for i in numbers():
        if i == n: break
        sum += i
    return sum
```

sum_to() 函数的类型提示表明它接收整型值作为参数，并返回整型值。sum 变量也使用了 Python 3 语法：:int，表明它是一个整型值。sum_to() 函数不对 numbers() 函数取所有值，在

取了前几个值后，就通过 break 语句退出了。下面的日志展示了 numbers()创建值的方式。

```
>>> sum_to(5)
= 0
= 1
= 2
= 3
= 4
= 5
10
```

后面会讲到，Python 生成器函数的一些特点使得它在应用于简单函数时会出现一些小麻烦，例如一个生成器只能用一次，因此在使用 Python 的惰性生成器表达式时要小心。

2.4　用递归代替循环语句

函数式编程不依赖循环语句，也不产生跟踪循环状态的开销，而使用相对简单的递归语句。在一些语言中，代码中的递归会在编译阶段被编译器通过**尾调用优化**（tail call optimization，TCO）技术转换成循环语句。本节简单介绍一些递归用法，第 6 章会详细讲解递归技术。

接下来介绍如何通过遍历测试一个数是否为质数。质数是只能被 1 和本身整除的自然数。我们可以定义一个简单的低性能算法，检查 2 到这个数之间的所有自然数能否整除它。该算法的优点是简单，可以作为**欧拉计划**中问题的解法。如果你对求解该问题的高性能算法感兴趣，可以参考**米勒–拉宾**素性检验算法。

我们使用“互质”表示两个数除了 1 之外没有其他公约数，比如 2 和 3 是互质的，而 6 和 9 不是互质的，因为它们除了 1 之外，还有公约数 3。

这样就可以把测试一个数是否为质数，转换为下面这个问题：自然数 n 是否与自然数 p 的任意取值互质，其中 $p^2 < n$，或者简化成：n 是否与所有满足条件 $2 \leqslant p^2 < n$ 的数 p 互质？

不妨把上面的陈述转换为如下数学公式。

$$\text{prime}(n) = \forall x[(2 \leqslant x < 1+\sqrt{n}) \wedge (n \mod x) \neq 0)]$$

对应的 Python 表达式如下。

```
not any(n % p == 0 for p in range(2,int(math.sqrt(n)) + 1))
```

从数学公式转换而来的更准确的写法是 all(n % p != 0 ...)，但它需要对所有 p 的可能取值严格求值，而上面的 not any 版本会在碰到第一个 True 值的时候结束计算。

这个简单的表达式中包含一个 for 循环，所以不是纯粹的、无状态的函数式编程风格。我们可以把它转换成处理一个集合的函数：自然数 n 是否与 $[2,1+\sqrt{n})$ 内的所有数互质？其中符号[)

代表半开区间：下限在范围内，上限不在范围内，与 Python 函数 `range()`的取值方式一致。由于我们只在自然数范围内讨论问题，所以平方根的值会自动转换为整数。

因此可以如下定义测试质数的公式。

$$\text{prime}(n) = \neg\text{coprime}(n, [2, 1+\sqrt{n}))\quad n > 1$$

基于一组数值定义函数时，集合为空是基础形式，递归将非空集合处理为：对集合中一个数的处理结果加上对剩余数值集合的处理结果，类似于下面的公式。

$$\text{coprime}(n,[a,b)) = \begin{cases} \text{True} & a = b \\ n \mod a \neq 0 \wedge \text{coprime}(n,[a+1,b)) & a < b \end{cases}$$

该公式通过判断以下两种情况给出测试结果。

- 当范围为空时，$a = b$，对类似于 coprime(131071, [363, 363))的表达式求值。由于范围内不含可用取值，所以返回 `True`。
- 当范围非空时，对类似于 coprime(131071, [2,363))的表达式求值。在这种情况下，表达式会分解为$(131071 \mod 2) \neq 0 \wedge \text{coprime}(131071, [3,363))$，其中第一部分的返回值为 `True`，所以需要递归对第二部分求值。

作为练习，读者可以尝试将上面的递归算法由向上缩减范围改为向下缩减范围，即在递归形式中用`[a, b-1)`代替`[a+1, b)`。

也许有人认为空区间应该是 a≥b 而非 `a = b`，其实是不必要的。由于 a 每次增加 1，可以保证 a≤b 始终成立，并不会因某种差错导致 a 越过 b。所以无须为空区间设置过多限制条件。

实现前面定义的质数的 Python 代码如下。

```
def isprimer(n: int) -> bool:
    def isprime(k: int, coprime: int) -> bool:
        """Is k relatively prime to the value coprime?"""
        if k < coprime*coprime: return True
        if k % coprime == 0: return False
        return isprime(k, coprime+2)
    if n < 2: return False
    if n == 2: return True
    if n % 2 == 0: return False
    return isprime(n, 3)
```

上面的代码使用了递归定义的函数 `isprimer()`，接收类型为整数的参数 `n`，返回布尔值。

半开区间$(2, 1+\sqrt{n})$的下限参数 a 改名为 `coprime` 以更好地表达其作用：通过改变它来缩小测试范围。这里的基础形式是 `n < coprime * coprime`，此时可以保证从 `coprime` 到 `1 + math.sqrt(n)`为空。

其中将非严格的 and 操作替换成一条单独的 if 测试语句：if n % coprime == 0。最后的 return 语句使用新的 coprime 参数对函数进行递归调用。

由于递归发生在函数尾部，所以这是一个**尾递归**的例子。

函数中增加了对待测试自然数的限制条件：大于 2 的奇数。由于 2 是唯一的偶质数，所以大于 2 的偶数显然不是质数，无须测试。

这个示例的重点是递归函数中的两种情况都很简单，用待测试值作为内部 isprime()函数的参数，可以让我们在递归调用函数时，通过改变参数值来确保取值范围不断缩小。

虽然这样的算法通常简洁明了，但在 Python 中使用这样的算法要注意下面两个问题。

- Python 对嵌套递归的深度有限制，不能随意定义基础形式。
- Python 的编译器没有尾递归功能。

递归深度上限默认为 1000，对于多数算法来说足够了。也可通过 sys.setrecursionlimit()函数修改这个值，但不建议把它改得过大，否则算法使用的内存会超过操作系统的物理内存，进而导致 Python 解释器崩溃。

当用 isprime()函数测试大于 1 000 000 的数时就会超过递归上限。即使我们优化算法，用它只测试质数，而非所有自然数，这个算法仍然只能测试第 1000 个质数（即 7919）的平方（即 62 710 561）以下的自然数。

有些函数式语言可以优化类似于 isprime()这样的函数，把 isprimer(n, coprime+1)这样的递归调用转换为低开销循环语句。但这种优化往往会弄乱调用栈，难以调试优化过的程序。Python 不进行此类优化，而为了保证清晰与简洁，牺牲了性能和内存使用，这也意味着我们必须手动进行优化。

在 Python 中，如果用生成器表达式代替递归函数，实际效果相当于手动进行了尾调用优化，从而不必依赖其他函数式语言中编译器的优化能力。

使用生成器表达式完成尾调用优化的代码示例如下。

```
def isprimei(n: int) -> bool:
    """Is n prime?

    >>> isprimei(2)
    True
    >>> tuple( isprimei(x) for x in range(3,11) )
    (True, False, True, False, True, False, False, False)
    """
    if n < 2:
        return False
    if n == 2:
```

```
        return True
    if n % 2 == 0:
        return False
    for i in range(3, 1+int(math.sqrt(n)), 2):
        if n % i == 0:
            return False
    return True
```

该函数满足函数式编程的许多要求，但用生成器表达式取代了单纯的递归调用。



在生成器表达式中常用 for 循环优化递归函数。

对于大的质数，上面的算法速度不算快，但对于合数，往往能很快给出结果。以 $M_{61} = 2^{61} - 1$ 为例，上述算法需要花费数分钟才能确定它是质数，毕竟整个过程测试了 1 518 500 249 个可能的因子。

2.5　函数类型系统

有些函数式语言（比如 Haskell 和 Scala）是静态编译的，通过类型声明确定函数及其参数的类型。为了具备 Python 式的灵活性，这些语言发展出了各自的类型匹配规则，从而能编写针对相近的一组类的抽象函数。

在面向对象的 Python 中，常用类继承代替复杂的函数类型匹配，利用 Python 的命名匹配规则将运算符与正确的方法关联。

由于 Python 语言非常灵活，并不需要编译式函数语言的类型匹配规则，甚至可以说，复杂的类型匹配规则不过是静态编译语言为了能编写抽象函数而采取的变通方法。Python 是动态语言，不需要采用这种变通方法。

Python 3 引入了类型提示功能，可以使用 mypy 等工具通过分析类型匹配发现潜在的问题。使用类型提示比通过类型测试（即 `assert isinstance(a, int)`）检查参数 a 的类型是否为整型值更好，因为 assert 语句增加了运行时的资源开销，而运行 mypy 验证提示是常规质量保证环节的一部分，通常与单元测试工具 pylint 等配合使用，以保证软件的正确性。

2.6　回到最初

文行至此，不难发现 Python 具备了函数式编程的绝大多数特征。实际上，这些函数式编程技术甚至已经广泛应用于面向对象编程中了。

作为特例，流畅的应用编程接口（API）很好体现了函数式编程的特质。在一个类中，只要花点时间在它的每个方法后面都加上 `return self()`，就可以编写出如下所示的代码：

```
some_object.foo().bar().yet_more()
```

或者把紧密相关的几个函数写成下面的形式：

```
yet_more(bar(foo(some_object)))
```

也就是是把传统面向对象风格的后缀式语法改为偏函数式的前缀式语法。两种写法均可用于 Python，不过对于有特殊意义的方法，主要用前缀形式，例如 `len()`函数实际上是通过 `class.__len__()`方法实现的。

当然，前面这些类的实现仍然可能包含有状态的对象，但即便如此，视角的微小改变也有助于我们在编程实践中灵活运用函数式方法，编写出简洁明了的程序代码。

再次强调，使用函数式编程并不意味着命令式编程存在某些严重缺陷，或者函数式编程能提供高新技术。函数式编程的精髓在于改变视角，这种改变往往有助于更好地编码。

2.7 几个高级概念

本书之后会讨论函数式编程的一些相关高级概念，这些概念在纯粹的函数式语言中是不可或缺的。由于 Python 并非纯函数式语言，而是一种混合式函数编程方法，所以不需要深入研究这些概念。

这部分讨论对那些熟悉函数式语言（例如 Haskell）的 Python 初学者特别有用，因为 Python 处理这些问题的方式与其他语言不同。很多时候，我们会采用命令式编程的方法解决问题，而不局限于函数式方法。

这些概念如下所示。

- **引用透明性**：在编译语言中，为了保证惰性求值和多种优化方法的正确性，需要确保通向同一对象有多条路径。在 Python 中，这一点并不是特别重要，因为 Python 没有编译阶段优化。
- **柯里化**：类型系统通过柯里化技术把多参数函数转化为单参数函数，第 11 章将深入讨论这个问题。
- **Monad**：将一系列操作灵活串联起来，构成一个纯函数。在某些场景中，可使用命令式 Python 代码达到相同的效果。也可以借助 Python 库 `PyMonad` 构造 Monad，第 14 章将深入讨论这个问题。

2.8 小结

本章阐述了函数式编程范式的几个核心特征。首先介绍了头等对象和高阶函数，其关键在于函数可以使用其他函数作为参数，或者返回另一个函数。当函数变成其他编程的“对象”时，就

可以编写出既灵活又具备抽象能力的算法了。

在 Python 等面向对象的语言中，不可变数据结构似乎有些另类，但当我们从函数式编程的角度思考时，就会发现状态变化是许多令人困惑的问题的根源。使用不可变数据结构有助于我们拨云见日，直抵问题的核心。

Python 使用严格求值策略：按照从左向右的顺序对语句中的表达式求值。但在处理逻辑运算符，比如 `or`、`and` 和 `if-else` 时，Python 又是非严格的，依据已知部分的值确定是否对未知部分求值。同样，生成器函数也不是严格求值的。这两种求值策略也称“积极求值”和“惰性求值”。总体而言，Python 使用积极求值策略，但可以通过生成器函数实现惰性求值。

函数式语言通常使用递归代替循环语句，但 Python 对此有一些限制。由于调用栈长度的限制，以及缺乏优化编译器，我们需要手动优化递归函数，第 6 章会详细讨论该话题。

大多数函数式语言都有各自复杂的类型系统，但我们使用 Python 的动态类型系统。虽然有时候这意味着必须手动转换类型，或者创建专门的类来应对复杂情形，但绝大多数时候，利用 Python 现有的规则就可以很好地解决问题。

下一章将介绍纯函数的一些核心概念，以及如何使用 Python 的内置数据结构实现这些概念。在此基础上，会介绍 Python 提供的高阶函数，以及自定义高阶函数的方法。

第 3 章

函数、迭代器和生成器

函数式编程的核心是使用纯函数将定义域中的值映射到值域。纯函数没有副作用，在 Python 中易于实现。

避免副作用意味着减少对通过变量赋值维护计算状态的依赖。我们不可能将赋值语句从 Python 中剔除，但可以减少对有状态对象的依赖，即需要从 Python 的内置数据结构中选择那些不依赖状态操作的数据类型。

本章将从函数视角介绍 Python 的如下特性。

- 无副作用的纯函数。
- 函数充当对象，用作参数或者返回值。
- 以面向对象的后缀方式和前缀方式使用 Python 字符串。
- 使用元组和命名元组创建无状态对象。
- 将可迭代集合作为函数式编程的首选设计工具。

生成器和生成器表达式是处理集合对象的重要工具，本章将详细介绍它们。第 2 章提到使用递归完全代替生成器表达式会有一些问题，包括嵌套递归深度限制，以及 Python 不会自动进行尾调用优化，所以需要使用生成器表达式来手动优化递归。

我们将使用生成器表达式完成如下工作：

- 转换
- 重构
- 进行复杂计算

本章将探讨 Python 的内置集合类型，以及在函数式范式下使用这些集合的方法，或许这将改变我们使用列表、字典和集合的方法。在函数式编程中，我们将侧重于使用 Python 的元组和其他不可变集合类型。下一章将详细介绍如何以函数式的方法使用特殊集合。

3.1 编写纯函数

没有副作用的函数符合在数学中函数的纯粹定义：变量不会在全局范围内发生变化，避免使用 global 语句基本可以达到要求。为了达到纯粹，则要求避免函数改变可变对象的状态。

下面是一个纯函数的例子：

```
def m(n: int) -> int:
    return 2 ** n - 1
```

返回结果仅与参数 n 的值有关，既没有改变全局变量，也没有更新任何可变数据结构。

任何（通过自由变量）对 Python 全局命名空间中值的引用都可以用参数实现，通常这类操作都很简单。下面是一个使用自由变量的例子：

```
def some_function(a: float, b: float, t: float) -> float:
    return a + b * t + global_adjustment
```

可以将上面的 global_adjustment 变量变为函数的参数，然后修改所有使用这个函数的地方。在一个复杂应用中，这么做会引发一系列变动。对全局变量的引用在函数体中表现为自由变量。

Python 的许多内置对象都带有状态，例如文件类以及与文件相关的对象，都是常用的有状态对象。可以把 Python 中的大多数有状态对象看作上下文管理器，虽然很多开发者不使用上下文管理器，但实际上很多对象都实现了它要求的接口。少数有状态对象没有完全实现上下文管理器的接口，但往往实现了 close()方法。可以通过 contextlib.closing()函数为这些对象实现对应的上下文管理接口。

除非是小程序，否则我们难以去除所有的 Python 有状态对象。因此，必须在发挥函数式优势与管理状态中间寻找平衡。为了达到目标，应尽量用 with 语句将有状态对象严格限制在一定的作用域内。



尽可能把文件对象放在 with 语句中。

尽量避免使用全局文件对象和全局数据库连接，以避免相关状态问题。全局文件对象是处理文件的常规方式，例如下面的命令片断所示的函数：

```
def open(iname: str, oname: str):
    global ifile, ofile
    ifile = open(iname, "r")
    ofile = open(oname, "w")
```

在这个场景中，其他函数可以随意使用 ifile 变量和 ofile 变量，而这两个变量不仅是全

局的，还始终保持打开的状态。

这样的设计不符合函数式编程的要求，应尽量避免。文件应作为函数的参数，打开的文件应嵌在 with 语句中，以确保能正确处理其状态。将全局变量改为普通参数非常重要，会使文件操作更易辨识。

该原则也适用于数据库，应把数据库连接对象作为应用程序中函数的普通参数。有些流行的 Web 框架把数据库连接设计成全局的，使得整个应用程序都可以使用数据库的某些功能。这种透明性隐藏了 Web 操作对数据库的依赖，使得单元测试变得非常复杂。但单一的全局数据库连接并不能很好地支持多线程 Web 服务，使用连接池往往更好。这也说明了总体使用函数式设计，辅以一些严格限制的状态对象，是处理这种情况的理想解决方案。

3.2 函数作为头等对象

之前讲过 Python 函数是头等对象，也是包含一些属性的普通对象。关于函数对象可用的特殊方法，可以参阅 Python 语言参考手册。与获取对象属性的方法相同，通过__doc__和__name__属性可以获得函数的 docstring 和名称，还可以通过__code__属性获得函数体。在编译语言中，实现这类自省机制相对复杂，因为要考虑如何保留源代码信息，但对于 Python 来说这很简单。

函数可以赋给变量，可以作为参数传递，还可以作为其他函数的返回值。运用这些技术可以轻松编写高阶函数。

由于函数已经是对象了，所以 Python 具备函数式语言的许多要素。另外，由于函数是普通的对象，可以通过可调用对象创建函数，甚至可以将可调用类视为高阶函数。在定义可调用对象的__init__()方法时要尽量避免在其中设置有状态的类变量。通常的做法是利用策略模式定义__init__()方法。

一个按照策略模式创建的类使用另一个对象为其提供算法，或者部分提供算法，这样就可以在运行时动态注入算法，而不必将具体逻辑固定在类内部。

在可调用对象中嵌入策略对象的示例如下：

```
from typing import Callable
class Mersenne1:
    def __init__(self, algorithm: Callable[[int], int]) -> None:
        self.pow2 = algorithm
    def __call__(self, arg: int) ->int:
        return self.pow2(arg) - 1
```

这个类使用__init__()方法将另一个函数 algorithm 的引用保存在 self.pow2 中。上面没有创建任何有状态的变量，self.pow2 不应发生变化。这里参数 algorithm 的类型标示是 Callable[[int], int]，表示该函数的输入和输出都是整型值。

这个函数根据策略对象提供的算法计算 2 的给定次方，可以作为类构造参数的 3 个备选算法如下所示：

```
def shifty(b: int) -> int:
    return 1 << b

def multy(b: int) -> int:
    if b == 0: return 1
    return 2 * multy(b - 1)

def faster(b: int) -> int:
    if b == 0: return 1
    if b % 2 == 1: return 2 * faster(b - 1)
    t = faster(b // 2)
    return t * t
```

`shifty()`函数通过左移位（bit）计算 2 的次方值。`multy()`函数利用简单的递归乘法计算。`faster()`函数利用分治策略，只需要执行 $\log_2(b)$次乘法运算，而非 `b` 次。

以上3个函数的签名完全一样：`Callable[[int], int]`，与`Mersenne1.__init__(self)`方法中的参数 `algorithm` 匹配。

下面就可以根据不同的算法创建 `Mersenne1` 类的实例了，如下所示：

```
m1s = Mersenne1(shifty)
m1m = Mersenne1(multy)
m1f = Mersenne1(faster)
```

这样就利用不同的算法，定义不同的函数，计算得到了相同的结果。



Python 可以计算 $M_{89}=2^{89}-1$，这还未接近其嵌套递归深度限制。这是一个很大的质数，有 27 位。

3.3　使用字符串

Python 的字符串是不可变的，所以非常适合函数式编程。Python 的 `string` 模块所含的方法都会创建新的字符串作为结果，这些方法都是没有副作用的纯函数。

字符串方法采用后缀写法，而大多数普通函数采用前缀写法，这导致字符串操作和普通函数运算混合在一起时，代码不易读。例如在表达式 `len(variable.title())`中，`title()`采用后缀写法，而 `len()`函数采用前缀写法。

从网页上爬取数据时，经常会用到清洗函数，对原始字符串执行一系列转换，去掉其中的标点符号，最后返回一个数值对象供其他函数使用，这时前缀写法和后缀写法就会混合在一起。

如下所示：

```
from decimal import *
from typing import Text, Optional
def clean_decimal(text: Text) -> Optional[Text]:
    if text is None: return None
    return Decimal(
        text.replace("$", "").replace(",", ""))
```

这个函数对原始字符串进行了两次替换，分别移除其中的$和逗号，替换结果作为 Decimal 类构造函数的参数，返回相应的数值对象。如果输入值是 None，则返回原值，所以使用了类型标示 Optional。

为了使语法更统一，可以自定义前缀式的字符串替换函数，如下所示：

```
def replace(str: Text, a: Text, b: Text) -> Text:
    return str.replace(a, b)
```

现在可以使用统一的前缀式写法了：Decimal(replace(replace(text, "$", ""), ",", ""))。对比原来的前缀–后缀混合写法，如此修改后，一致性似乎没有明显提升，这是一个一致性应用欠妥的例子。

更好的做法是重新定义一个语义清晰的前缀式函数来剔除字符串中的标点符号，如下所示：

```
def remove(str: Text, chars: Text) -> Text:
    if chars:
        return remove(
            str.replace(chars[0], ""),
            chars[1:]
        )
    return str
```

该函数递归地剔除 chars 变量中的每个字符，因此可以把原来分别剔除 $ 和逗号的函数改写成更清晰的形式：Decimal(remove(text, "$,"))。

3.4 使用元组和命名元组

Python 的元组也是不可变对象，因此很适合函数式编程。元组数据结构可用的方法不多，大多数处理都是通过前缀式语法完成的。元组应用广泛，主要用于元组列表、嵌套元组和元组生成器。

命名元组这个类为元组增加了一个重要特征：通过属性名称而非序号引用属性。我们可以使用命名元组创建数据累积式的对象，这样就能基于无状态对象创建纯函数，同时保持数据以对象形式有序地组织在一起了。

后续将主要使用元组和命名元组处理集合数据。对于单个或者两个独立值，可以采用函数的

命名参数；而当处理集合数据时，则需要使用可迭代元组或者可迭代命名元组。

使用元组还是命名元组取决于具体场景中使用哪种数据结构更方便，例如可以把包含红、绿、蓝三种颜色的数据抽象为形如(number, number, number)的三元组，但这个定义没有红、绿、蓝的顺序。有几种方法可以让元组的结构更清晰。

可以通过下面这个函数从该三元组中抽取数据来表明其结构。

```
red = lambda color: color[0]
green = lambda color: color[1]
blue = lambda color: color[2]
```

对于元组 item，可以使用 red(item)从中选出红色成分，并添加更正式的类型标示。

```
from typing import Tuple, Callable
RGB = Tuple[int, int, int]
red: Callable[[RGB], int] = lambda color: color[0]
```

这里定义了代表三元组的新类型 RGB。red 的类型标示是 Callable[[RGB], int]，表示它是一个函数，输入参数类型为 RGB，返回一个整型值。

或者使用旧式命名元组来定义：

```
from collections import namedtutple
Color = namedtuple("Color", ("red", "green", "blue", "name"))
```

更好的方法是使用 typing 模块中的 NamedTuple 类：

```
from typing import NamedTuple
class Color(NamedTuple):
    """An RGB color."""
    red: int
    green: int
    blue: int
    name: str
```

Color 类定义了元组中每个位置的名字和类型标示，在保持性能优势和不可变性的基础上，还可以利用 mypy 校验该类型变量的使用方法是否恰当。

通过上面两种方法提取红色时，可以用 item.red 代替 red(item)，它们都比 item[0]可读性好。

元组在函数式编程中的应用侧重于可迭代元组设计模式。之后会详细介绍几种可迭代元组技术，第 7 章将介绍命名元组技术。

3.4.1 使用生成器表达式

前面给出了一些生成器表达式的示例，稍后会给出更多生成器表达式，本节将介绍关于生成

器的一些高级技术。

生成器表达式常用于通过列表推导或者字典推导生成列表或者字典字面量。以`[x**2 for x in range(10)]`为例，它是一个列表推导，也称列表展示。列表展示是生成器表达式的一种用法，集合（列表或字典）展示包含闭合的字面语法，这里是用列表的字面语法`[]`包裹生成器`[x**2 for x in range(10)]`，表示这是一个列表推导，返回一个由其中包含的生成器表达式 `x**2 for x in range(10)`生成的列表对象。本节主要介绍生成器表达式，不涉及列表对象。

集合对象和生成器表达式都是可迭代的，因此许多行为相似，但稍后会看到，它们并不等同。使用展示的缺点是会生成一个（可能比较大的）集合对象，而生成器表达式是惰性的，按需创建值，因而可提升性能。

对于生成器表达式，需要说明两点：

- 除了被一些特殊函数（例如 `len()`等需要知道整个集合大小的函数）使用时，生成器的行为与序列类似；
- 生成器只能使用一次，可将使用完的生成器视为空的。

下面是一个后续示例会用到的生成器函数：

```
def pfactorsl(x: int) -> Iterator[int]:
    if x % 2 == 0:
        yield 2
        if x // 2 > 1:
            yield from pfactorsl(x // 2)
        return
    for i in range(3, int(math.sqrt(x) + .5) + 1, 2):
        if x % i == 0:
            yield i
            if x // i > 1:
                yield from pfactorsl(x // i)
            return
    yield x
```

该函数计算输入值的所有质因数。如果输入值 x 是偶数，则返回 2，然后递归计算出 $x/2$ 的所有质因数。

对于奇数，从 3 开始依次验证每个奇数是不是输入值的因数，如果是，返回该因数 i，然后递归生成 $x \div i$ 的所有质因数。

如果输入值没有质因数，它本身一定是质数，会返回它自身。

这里 2 作为一个特殊值，每次可以将迭代次数减半，因为 2 以外的所有质数都是奇数。

除了递归，上面的函数还使用了 `for` 循环，这样就可以处理质因数个数多达 1000 的输入值了（以 2^{1000} 为例，它有 300 位长，有 1000 个质因数）。由于没有在 `for` 循环外使用循环变量 `i`，因此当修改循环体时，它有状态的特点并不会带来麻烦。

这个例子展示了如何手动实现尾递归优化，用循环代替了从 3 到 $\sqrt{x}$ 的递归求值，避免了深度嵌套的递归调用。

由于函数是可迭代的，用 `yield from` 语句从递归调用中计算出所有值，返回给调用者。



在递归生成器函数中要谨慎使用 `return` 语句，不要用下面的命令行：

```
Return recursive_iter(args)
```

它只返回生成器对象，而不对函数求值并返回结果，下面两种写法可行：

```
for result in recursive_iter(args):
    yield result
```

或者

```
yield from recursive_iter(args)
```

上面函数的纯递归版本如下：

```
def pfactorsr(x: int) -> Iterator[int]:
    def factor_n(x: int, n: int) -> Iterator[int]:
        if n*n > x:
            yield x
            return
        if x % n == 0:
            yield n
            if x//n > 1:
                yield from factor_n(x//n, n)
        else:
            yield from factor_n(x, n+2)
        if x % 2 == 0:
            yield 2
            if x//2 > 1:
                yield from pfactorsr(x//2)
            return
        yield from factor_n(x, 3)
```

首先定义内部递归函数 `factor_n()` 测试 $3 \leqslant n \leqslant \sqrt{x}$ 范围内 n 是否为 x 的因数，如果待测的 n 不在此范围内，说明 x 是质数。否则检查 n 是否是 x 的因数，如果是，返回 n，并递归生成 $\frac{x}{n}$ 的所有质因数，否则递归测试 n+2 是否是 x 的因数，这里需要递归测试 $(n+2, n+2+2, n+2+2+2, \cdots)$ 中的所有值，虽然它在写法上比前面的 `for` 循环版本简单，但由于 Python 栈长度限制，无法处

理因数个数超过 1000 的情况。

外层函数处理边界情况。与其他质数处理算法一样，这里将 2 作为特殊情形，对于偶数，首先返回 2，然后递归生成 $x \div 2$ 的质因数。其他质因数一定是不小于 3 的奇数，所以从 3 开始用 `factor_n()` 函数依次测试每个可能的解。



纯递归函数最多能处理约 4 000 000 个输入值。超过这个值，计算会由于超出 Python 的嵌套递归深度上限而失败。

3.4.2　生成器的局限

生成器表达式和生成器函数是有局限的，如下所示：

```
>>> from ch02_ex4 import *
>>> pfactorsl(1560)
<generator object pfactorsl at 0x1007b74b0>
>>> list(pfactorsl(1560))
[2, 2, 2, 3, 5, 13]
>>> len(pfactorsl(1560))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'generator' has no len()
```

第一个例子表明生成器函数不是严格求值的，是惰性的，在外部对其求值之前，并不进行实际的计算。这不是局限，而是将生成器表达式用于 Python 函数式编程的原因。

第二个例子通过实例化生成器函数得到一个列表对象，便于观察输出和编写单元测试用例。

第三个例子体现了生成器函数的一个局限：不能使用 `len()` 函数求其长度。因为生成器是惰性的，所以在所有值都取完之前，无法知道值的总数。

生成器函数的另一个局限是只能使用一次，如下所示：

```
>>> result = pfactorsl(1560)
>>> sum(result)
27
>>> sum(result)
0
```

第一次使用 `sum()` 函数对生成器求值后，第二次再使用时生成器为空了，表明生成器只能用一次。

Python 中的生成器是有生命周期的，从函数式编程的角度看这不是特别理想。

可以使用 `itertools.tee()` 函数弥补只能用一次的限制，第 8 章会详细介绍这一点。下面是一个简单的示例：

```
import itertools
from typing import Iterable, Any
def limits(iterable: Iterable[Any]) -> Any:
    max_tee, min_tee = itertools.tee(iterable, 2)
    return max(max_tee), min(min_tee)
```

这为作为参数传入的生成器表达式创建了两个副本：`max_tee()`和 `min_tee()`，保持原始生成器不变，借助这个功能，可以非常灵活地把函数组合在一起。通过对两个副本求值，得到生成器计算结果的最大值和最小值。

可迭代对象中的值取完后，将不再生成任何值。当需要对生成器进行多种归约时，例如求和、计数、求最大值或最小值等，请注意，设计算法时必须考虑到只能用一次这个特点。

3.4.3 组合生成器表达式

函数式编程的核心特征是：通过灵活地组合生成器表达式和生成器函数描述复杂的处理流程。下面介绍使用生成器表达式时几种常用的组合方法。

第一种方法是通过创建复合函数来组合生成器。假设有个生成器`(f(x) for x in range())`，如果需要计算 `g(f(x))`，可用以下几种方法来组合它们。

可以把原来的生成器表达式改成如下形式：

```
g_f_x = (g(f(x)) for x in range())
```

这种方式从技术上说是正确的，但违背了复用原则：没有复用已有的表达式，而是改写了。

也可将一个表达式嵌入另一个表达式中，如下所示：

```
g_f_x = (g(y) for y in (f(x) for x in range()))
```

这样就可以方便地进行变量替换了。可通过如下命令实现复用：

```
f_x = (f(x) for x in range())
g_f_x = (g(y) for y in f_x)
```

这种实现方法的优点是无须改写原来的表达式`(f(x) for x in range())`，只需把它赋给一个变量即可。

组合后的表达式仍然是生成器表达式，并且是惰性的。当从 `g_f_x` 中取出一个值时，它会从 `f_x` 中取一个值，`f_x` 再从 `range()`函数中取一个值。

3.5 使用生成器函数清洗原始数据

EDA 中经常需要清洗原始数据，通过执行一系列标量函数，把输入数据转换为可用的数据集合。

下面介绍一个简化的数据集合，这是 EDA 领域中常用于演示数据处理技术的数据集：安斯库姆四重奏，名字来自 F. J. Anscombe 于 1973 年在《美国统计学家》杂志上发表的文章“Graphs in Statistical Analysis”。该数据集的前几行如下所示：

```
Anscombe's quartet
I  II  III  IV
x  y  x  y  x  y  x  y
10.0  8.04  10.0  9.14       10.0  7.46  8.0  6.58
8.0      6.95  8.0  8.14  8.0  6.77  8.0  5.76
13.0  7.58  13.0  8.74  13.0  12.74  8.0  7.71
```

csv 模块无法直接解析这样的数据格式，需要先解码才能从文件中抽取有用的信息。由于数据之间都是用 Tab 分隔的，可以使用 csv.reader() 函数处理每一行数据。首先定义一个数据迭代器，如下所示：

```
import csv
from typing import IO, Iterator, List, Text, Union, Iterable
def row_iter(source: IO) -> Iterator[List[Text]]:
    return csv.reader(source, delimiter="\t")
```

这里只是简单地把文件对象包裹在 csv.reader() 函数中，生成一个行迭代器。类型模块为文件对象提供了方便的定义：IO。csv.reader() 负责迭代处理所有行，每一行是一个文本值列表。额外添加一个类型定义 Row = List[Text] 会使之更明确。

在如下所示的上下文中使用 row_iter() 函数：

```
with open("Anscombe.txt") as source:
    print(list(row_iter(source)))
```

虽然确实包含了有用的数据，但返回结果的前 3 行并不是数据，如下所示：

```
[["Anscombe's quartet"],
 ['I', 'II', 'III', 'IV'],
 ['x', 'y', 'x', 'y', 'x', 'y', 'x', 'y'],
```

需要过滤掉这些非数据的行。下面的函数会移除迭代器的前 3 行，返回包含剩余行的迭代器。

```
def head_split_fixed(
        row_iter: Iterator[List[Text]]
    ) -> Iterator[List[Text]]:
    title = next(row_iter)
    assert (len(title) == 1
        and title[0] == "Anscombe's quartet")
    heading = next(row_iter)
    assert (len(heading) == 4
        and heading == ['I', 'II', 'III', 'IV'])
    columns = next(row_iter)
    assert (len(columns) == 8
        and columns == ['x','y', 'x','y', 'x','y', 'x','y'])
    return row_iter
```

该函数会移除可迭代对象的前 3 行，并检查剩余每一行是否符合预期，如果不相符，说明文件已损坏，或者这并不是要处理的文件。

由于 `row_iter()` 和 `head_split_fixed()` 函数都使用可迭代对象作为输入参数，可以将它们合并，如下所示：

```
with open("Anscombe.txt") as source:
    print(list(head_split_fixed(row_iter(source))))
```

将一个迭代器的处理结果作为参数传递给另一个迭代器，实际上这是一个复合函数。当然，整个数据清洗并没有到此结束，还需要将字符串转换为浮点数，然后将每行中 4 个并行序列的数据分开。

使用高阶函数（如 `map()` 和 `filter()`）来进行最终的转换和数据抽取会比较简单，第 5 章将详细介绍。

3.6 使用列表、字典和 set

Python 的序列对象（例如列表）是可迭代的，当然还有其他一些特点，可以把它们看作实例化的可迭代对象。在前面的例子中，使用了 `tuple()` 函数将生成器表达式或生成器函数的输出收集到单个元组对象中。也可以将序列实例化，以生成列表对象。

Python 的列表推导提供了实例化生成器的一种简单方法：添加方括号 `[]`。生成器表达式和列表推导基本没有区别，不用纠结于生成器表达式和使用了生成器表达式的列表推导的区别。

相关示例如下：

```
>>> range(10)
range(0, 10)
>>> [range(10)]
[range(0, 10)]
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

第一个例子是 `range` 对象，是一种生成器函数。它是惰性的，所以创建后不生成任何数值。



`range(10)` 是惰性的，仅在需要迭代所有值时才会执行 10 次求值。

第二个例子是一个包含生成器函数的列表，`[]` 语法创建了一个包含 `range` 对象的列表字面量，不对迭代器生成的值执行任何求值操作。

第三个例子演示了如何通过把生成器函数包含在生成器表达式中来创建列表推导。生成器表达式 `x for x in range(10)` 对生成器函数 `range(10)` 求值，返回结果保存在列表对象里。

另外，还可以使用 `list()` 函数，基于可迭代对象或者生成器表达式创建列表。该方法同样适用于 `set()`、`tuple()` 和 `dict()`。



`list(range(10))` 函数会对内部的生成器函数求值，而 `[range(10)]` 列表字面量不会。

在 Python 中，可以通过 `[]` 和 `{}` 这样的简单形式创建列表、字典和 set，而要实例化元组对象，必须使用 `tuple()` 函数。为了保持代码风格一致，建议统一使用函数名，如 `list()`、`tuple()` 和 `set()` 实例化对象。

前面数据清洗的例子使用一个复合函数生成了一个包含 4 个元组的列表，该函数如下所示：

```
with open("Anscombe.txt") as source:
    data = head_split_fixed(row_iter(source))
    print(list(data))
```

将复合函数的返回结果赋给变量 `data`，`data` 如下所示：

```
[['10.0', '8.04', '10.0', '9.14', '10.0', '7.46', '8.0', '6.58'],
 ['8.0', '6.95', '8.0', '8.14', '8.0', '6.77', '8.0', '5.76'],
 ...
 ['5.0', '5.68', '5.0', '4.74', '5.0', '5.73', '8.0', '6.89']]
```

至此还需要些许处理才能使之达到可用状态。首先要从八元组中拆出 4 组数据，下面的函数实现了按列分组：

```
from typing import Tuple, cast

Pair = Tuple[str, str]
def series(
        n: int, row_iter: Iterable[List[Text]]
    ) -> Iterator[Pair]:
    for row in row_iter:
        yield cast(Pair, tuple(row[n * 2: n * 2 + 2]))
```

这个函数将相邻的两列分成一组，按照从 0 到 3 的顺序一共分成 4 组，用组内的两列生成一个 `tuple` 对象。其中 `cast()` 函数用于显式说明 mypy 工具返回的结果是一个元素为字符串的二元组。之所以要特别指明，是因为 mypy 等工具很难分析出 `tuple(row[n * 2: n * 2 + 2])` 从行集合中取出的是两个元素。

使用如下函数可以生成嵌套元组的集合：

```
with open("Anscombe.txt") as source:
    data = tuple(head_split_fixed(row_iter(source)))
    sample_I = tuple(series(0, data))
```

```
    sample_II = tuple(series(1, data))
    sample_III = tuple(series(2, data))
    sample_IV = tuple(series(3, data))
```

这里用 tuple()函数处理之前由 head_split_fixed()和 row_iter()方法组成的复合函数的返回值，生成的对象可供后续函数复用。如果这里没有把返回值实例化为一个元组，后面的处理函数只有第一个能获得实际输出数据，之后作为数据源的迭代器被置空，再访问它只会返回空结果。

series()将提取出的两个数据转换为 Pair 对象，然后用 tuple()函数实例化返回的嵌套元组序列，以便进一步处理各元素。

sample_I 中的数据如下所示：

```
(('10.0', '8.04'), ('8.0', '6.95'), ('13.0', '7.58'),
('9.0', '8.81'), ('11.0', '8.33'), ('14.0', '9.96'),
('6.0', '7.24'), ('4.0', '4.26'), ('12.0', '10.84'),
('7.0', '4.82'), ('5.0', '5.68'))
```

另外 3 个序列与上面的序列虽然值不同，但结构相同。

整个处理流程的最后一步是把得到的字符串转换为数值，然后计算各项统计数据。这一步中的转换通过 float()函数实现，该函数可以放在许多不同的地方。第 5 章将介绍几种不同的方案。

如下所示使用 float()函数：

```
mean = (
    sum(float(pair[1]) for pair in sample_I) / len(sample_I)
)
```

这样就算出了每个二元组中 y 的平均值。可以用下面的方法算出各个集合的统计值：

```
for subset in sample_I, sample_II, sample_III, sample_III:
    mean = (
        sum(float(pair[1]) for pair in subset) / len(subset)
    )
    print(mean)
```

至此就计算出了源数据库中每对数据的 y 的平均值，并且创建了一个以命名元组为元素的元组序列，用于描述源数据库中的数据。pair[1]这种写法可读性较差，第 7 章将介绍如何使用命名元组简化复杂元组结构中对数据项的引用。

为了降低内存占用、提高性能，建议尽量使用生成器表达式和函数，它们采用惰性（非严格）方式迭代对集合元素求值，按需进行计算。由于迭代器只能使用一次，有时必须将集合实例化为元组对象或者列表对象。实例化集合耗费内存和时间，所以只在必要的时候这么做。

熟悉 Clojure 的开发者不难发现，Python 的惰性生成器与 Clojure 的 lazy-seq 和 lazy-cat

函数是对等的，它们提供了相同的功能：开发者可以定义一个无限序列，仅在需要的时候从中取值。

3.6.1　使用状态映射

Python 提供了几个有状态的集合数据结构，包括字典类和 `collections` 模块中定义的映射数据结构。这些数据结构都是有状态的，使用的时候要小心。

就本书主题而言，映射主要有两个使用场景：将映射聚合在一起的有状态字典和不可变字典。本章的第一个例子演示了 `ElementTree.findall()`方法如何使用不可变字典。Python 没有提供简单易用的不可变映射数据结构，虽然 `collection.abc.Mapping` 类是不可变的，但不易使用，第 6 章将详细讨论这个话题。

在不考虑使用 `collections.abc.Mapping` 抽象类的情况下，可以采取一些简单的变通方法保证数据的不可变性质，包括映射变量（不妨叫它 `ns_map`）只出现在赋值语句左边一次，避免使用 `ns_map.update()`或者 `ns_map.pop()`方法，不在 `del` 语句中使用映射变量等。

有状态的字典数据结构主要有如下两个使用场景。

- 只创建一次，且不再更新。在这种情况下，通常使用 `dict` 类的散列键功能提高性能。通过 `dict(seqence)`函数，可以基于任何`(key, value)`组成的可迭代序列创建字典。
- 叠加式构造字典。使用它避免对列表对象进行实例化和排序。第 6 章会通过分析 `collections.Counter` 类讲解高级归约技术。另外，在记忆化技术中，也常用叠加式构造。第 16 章会详细介绍记忆化技术。

作为上面使用场景的第一个例子，“只创建不修改”这种方式主要适用于三段式处理应用：收集输入，创建一个字典对象，最后根据字典中的映射处理输入。这类场景的一个常见应用是在图像处理中用(R, G, B)元组指定颜色值。当使用 GIMP（GNU Image Manipulation Program）文件格式时，调色板定义如下所示：

```
GIMP Palette
Name: Small
Columns: 3
#
  0   0   0    Black
255 255 255    White
238  32  77    Red
28  172 120    Green
31  117 254    Blue
```

第 6 章将详细介绍解析这类文件的方法，这里只关注解析的结果。

首先，定义一个名为 `Color` 的命名元组：

```
from collections import namedtuple
Color = namedtuple("Color", ("red", "green", "blue", "name"))
```

接下来，假设我们用解析器生成了由 Color 对象组成的解析器，如果将其实例化，转换成的元组如下所示：

```
(Color(red=239, green=222, blue=205, name='Almond'),
 Color(red=205, green=149, blue=117, name='Antique Brass'),
 Color(red=253, green=217, blue=181, name='Apricot'),
 Color(red=197, green=227, blue=132, name='Yellow Green'),
 Color(red=255, green=174, blue=66, name='Yellow Orange'))
```

为了能根据颜色名称快速定位到颜色元组，需要基于该序列创建一个不可变字典。当然，这不是根据名称快速定位的唯一方法，后面会介绍其他方法。

从元组创建映射，需要使用之前讲过的 process(wrap(iterable))设计模式。创建颜色映射如下所示：

```
name_map = dict((c.name, c) for c in sequence)
```

其中序列变量是前面提到的 Color 对象，模式中的 wrap()步骤将每个 Color 对象转换为元组(c.name, c)，process()步骤使用 dict 初始化方法生成一个从名称到 Color 对象的映射。转换结果如下所示：

```
{'Caribbean Green': Color(red=28, green=211, blue=162,
 name='Caribbean Green'),
 'Peach': Color(red=255, green=207, blue=171, name='Peach'),
 'Blizzard Blue': Color(red=172, green=229, blue=238, name='Blizzard Blue'),
 etc.
}
```

由于字典中元素的顺序是随机的，所以运行结果中 Caribbean Green 可能不在第一位。

这样就完成转换了，生成的字典对象后续将多次用于从名称到(R, G, B)色值的转换过程中。由于字典在查找元素时首先将键值转换为散列值，所以查找速度会非常快。

3.6.2 使用 **bisect** 模块创建映射

前面的例子通过创建字典实现了从颜色名称到 Color 对象的快速映射。除此之外，还可以使用 bisect 模块来实现。使用 bisect 模块需要先创建一个有序对象，然后才能搜索。为了保持与字典映射结构兼容，可以使用 collections.abc.Mapping 作为基类。

字典映射使用散列值搜索元素，速度非常快，但需要消耗大量内存。相比而言，bisect 映射的搜索速度也很快，并且使用的内存较少。

下面定义一个静态映射类：

```
import bisect
from collections import Mapping
from typing import Iterable, Tuple, Any
class StaticMapping(Mapping):
    def __init__(self,
      iterable: Iterable[Tuple[Any, Any]]) -> None:
        self._data = tuple(iterable)
        self._keys = tuple(sorted(key for key, _ in self._data))
    def __getitem__(self, key):
        ix = bisect.bisect_left(self._keys, key)
        if ix != len(self._keys)
          and self._keys[ix] == key_:
            return self._data[ix][1]
        raise ValueError("{0!r} not found".format(key))
    def __iter__(self):
        return iter(self._keys)
    def __len__(self):
        return len(self._keys)
```

这个类从抽象超类 `collections.abc.Mapping` 派生而来，它定义了新的初始化方法，并增加了基类中没有的 3 个方法。`Tuple[Any, Any]`定义了一个代表普遍情形的二元组。

`__getitem__()`方法用 `bisect.bisect_left()`函数搜索集合内的所有键，如果找到了目标键，则返回相应结果。`__iter__()`方法返回超类需要的迭代器，`__len__()`方法与之类似，返回整个集合的长度。

或者参照 `collections.OrderedDict` 类的代码，把超类从 `MutableMapping` 改成 `Mapping`，并移除能修改状态的所有方法。8.4.1 节会介绍应保留或摒弃哪些方法。

更多相关信息，可访问 https://docs.python.org/3.3/library/collections.abc.html#collections-abstract-base-classes。

这个类并没有严格遵循函数式编程的原则，我们的目标是为大型应用提供一种尽量少使用有状态变量的方法，这个类保存了键值对的静态集合。作为优化手段，它实例化了两个对象。

通过实例化对象，使用这个类的应用能实现快速键查找。它的超类不允许更新对象，所以整个集合是无状态的。它的速度没有内置的字典快，但使用内存少。由于这个类的基类是 `Mapping`，可以想见这个类的实例不是用于存储过程状态的。

3.6.3　使用有状态的 set

Python 提供了多种有状态的集合，包括 set 集合。本书中 set 有两个主要的使用场景：用有状态的 set 收集数据，或者用不可变 set（`frozenset`）优化数据搜索算法。

基于可迭代对象创建 `frozenset` 跟通过 `fronzenset(some_iterable)`方法创建元组对

象的方法类似。创建的集合可以执行快捷 `in` 操作。整个处理过程是：收集数据，创建 set，用此 `frozenset` 处理其他数据。

有时会用一组颜色进行色键处理，也就是借助某个颜色生成的遮罩来合并两张图片。实践中，使用单一颜色的色键效果不理想，而使用一组相近的颜色效果会好很多，这时就需要判断一个图像文件的每个像素是否在色键集合中。在这种情况下，通常的做法是在处理目标文件之前，先把所有的色键颜色放入一个 `frozenset` 里。关于色键处理的更多信息，可参考维基百科词条 Chromakey。

对于映射（尤其是 `Counter` 类），使用记忆化数值往往能大幅提高算法性能。有些函数运用记忆化技术提高性能，因为它在域值和范围值之间做映射，而这恰好是映射数据结构的专长。另一些算法则在处理数据的过程中维护不断积累的记忆化数据集合，以此提高性能。

第 16 章会介绍记忆化的相关内容。

3.7　小结

本章重点介绍了如何编写没有副作用的纯函数。实现这一点并不难，因为 Python 要求在写不纯粹的函数时必须使用 `global` 语句。然后探讨了生成器函数，以及如何以其为基础进行函数式编程。还介绍了 Python 内置的集合类，以及如何在函数范式中使用它们。函数式编程整体上尽量避免使用有状态的变量，然而集合数据类型往往是有状态的，而且许多算法正是使用了集合有状态的特性，所以在利用 Python 的非函数特性时，务必考虑清楚。

接下来的两章主要讨论高阶函数：以函数为参数，或者返回另一个函数的函数。首先会介绍 Python 内置的高阶函数，然后会介绍自定义高阶函数的相关技术，以及 `itertools` 模块和 `functools` 模块中的高阶函数。

第4章

使用集合

Python提供了很多能处理集合的函数，可以用于序列（列表或元组）、set、映射，以及生成器表达式返回的可迭代对象。本章将从函数式编程的视角研究Python的集合处理函数。

首先介绍可迭代对象及相关的简单函数，然后研究几种使用递归函数和`for`循环处理可迭代对象和序列的设计模式，最后讲解如何将标量函数应用于带有生成器表达式的集合。

本章将介绍如何使用下列函数处理集合：

- `any()`和`all()`
- `len()`、`sum()`和一些统计处理相关的高阶函数
- `zip()`以及一些与结构化或者平铺列表数据相关的技术
- `reversed()`
- `enumerate()`

前4个函数属于**归约函数**，它们将一个集合归约为单个值。其他3个函数`zip()`、`reversed()`和`enumerate()`属于映射函数，基于已有的集合生成新集合。下一章主要介绍映射和归约函数，它们可以使用附加的函数参数来定义具体的处理方法。

本章首先讲解如何使用生成器表达式处理数据，然后用多种集合级函数来演示它们如何简化迭代处理语法，最后介绍重新构建数据的几种方法。

下一章重点介绍如何使用高阶函数达到相同的目标。

4.1 函数分类概览

函数大体可分为以下两类。

- **标量函数**：作用于单个值，并返回单个值，例如`abs()`、`pow()`以及整个`math`模块中的函数都是标量函数。
- **集合函数**：作用于可迭代集合。

集合函数又可以细分为以下三类。

- **归约**：通过指定函数将集合内元素汇聚在一起，生成单个值作为结果。例如通过加法运算可以得到集合内数据的和。由于这类函数总是以集合为输入，返回单个累积值，所以也称为**累积函数**。
- **映射**：将标量函数作用于集合的每个元素，作为结果返回的集合与输入集合长度相同。
- **过滤**：将标量函数作用于集合的每个元素，保留其中一部分元素，舍弃另一部分元素，返回的集合是输入集合的子集。

本章将在此概念框架内介绍使用内置集合函数的方法。

4.2 使用可迭代对象

前面讲过在 Python 中处理集合时，经常使用 `for` 循环语句。当处理实例化了的集合数据（例如元组、列表、映射与 set）时，`for` 循环中包含了显式的状态管理。虽然这种用法不符合函数式编程的原则，但反映出它是 Python 的一种必要的优化手段。如果能保证状态管理只用于 `for` 语句对应的迭代器对象，就可以在不过分偏离纯粹的函数式编程原则的前提下利用这一语言特性。如果在 `for` 循环体外部使用循环变量，就背离了函数式编程的原则。

第 6 章将深入探讨该话题，这里只使用生成器简单给出该问题的一个例子。

`for` 循环迭代处理常用于 `unwrap(process(wrap(iterable)))`设计模式。`wrap()`函数将原始可迭代对象中的每个元素转换为一个二元组，其中第一个元素是用于排序的键或者其他数值，第二个元素是原来不可变的输入数据。接着基于前面打包的结果处理数据。最后使用 `unwrap()` 函数拆分处理后的元组，舍弃打包数值，保留处理结果。

在函数式语境下上述操作十分频繁，因此定义如下两个函数：

```
fst = lambda x: x[0]
snd = lambda x: x[1]
```

这两个函数从元组中分别取出第一个元素和第二个元素，在 `process()`函数和 `unwrap()` 函数中经常用到。

另一个常用的模式是 `wrap3(wrap2(wrap1()))`。这种情况下，我们从简单的元组开始，加入后续计算结果，打包成新的元组，从而形成更大也更复杂的元组。该设计模式的一个变体是基于源对象创建新的、更复杂的命名元组实例。这两种模式都属于**生长设计模式**。

生长设计模式的一个应用是处理经纬度数值。处理过程的第一步是将路径上的点`(lat, lon)`转换为路径段`(begin, end)`，计算结果表示为：`((lat, lon), (lat, lon))`。对集合中的每个元素，用 `fst(item)`获取起点值，用 `snd(item)`获取终点值。

稍后演示如何创建一个生成器函数遍历输入文件中的数据，将后面需要处理的原始数据保存到可迭代对象中。

获得原始数据后，将演示如何计算某段路径的半正矢距离。经 wrap(wrap(iterable())) 处理后，数据变成一个三元组序列：((lat, lon), (lat, lon), distance)，这样就可以计算最长距离、最短距离、外接矩形及其他数据汇总值了。

4.2.1　解析 XML 文件

首先通过解析一个 XML（Extensible Markup Language）文件获得原始经纬度数据。该过程将展示封装 Python 中不那么函数式的一些语言特征，来生成一组可迭代序列的方法。

借助 xml.etree 模块，解析后返回的 ElementTree 对象通过 findall()方法遍历处理后的数据。

待处理的目标数据的 XML 代码如下所示：

```
<Placemark><Point>
<coordinates>-76.33029518659048,
    37.54901619777347,0</coordinates>
</Point></Placemark>
```

该文件包含多个<Placemark>标签，每个这样的标签中又包含点和坐标结构体，这是包含地理位置信息的 KML（Keyhole Markup Language）文件的典型格式。

解析 XML 文件的方法可以抽象为两个层次，底层方法负责定位各种标签、属性值以及文档内容，上层方法负责从文本和属性值中抽取有用的对象。

底层处理方法如下所示：

```
import xml.etree.ElementTree as XML
from typing import Text, List, TextIO, Iterable
def row_iter_kml(file_obj: TextIO) -> Iterable[List[Text]]:
    ns_map = {
        "ns0": "http://www.opengis.net/kml/2.2",
        "ns1": "http://www.google.com/kml/ext/2.2"}
    path_to_points= ("./ns0:Document/ns0:Folder/ns0:Placemark/"
          "ns0:Point/ns0:coordinates")
    doc = XML.parse(file_obj)
    return (comma_split(Text(coordinates.text))
            for coordinates in
            doc.findall(path_to_points, ns_map))
```

函数以 with 语句中文件对象的文本为输入，返回结果是基于经纬度数值对创建的生成器，用于生成包含数据的列表对象。解析 XML 文件的过程中，该函数包含一个简单的静态字典对象和 ns_map 对象，提供作为搜索目标的 XML 标签的命名空间映射信息，字典对象则供负责处理

XML 文件的 ElementTree.findall()方法使用。

解析的主体是一个生成器函数，通过 doc.findall()方法定位一系列标签，这些标签作为 comma_split()函数的参数，把目标文本转换为一组以逗号分隔的序列值。

其中的 comma_split()是字符串 split()方法的函数版本，具体实现如下：

```
def comma_split(text: Text) -> List[Text]:
    return text.split(",")
```


通过将对象方法包装为前缀式函数，保证了语言风格的统一。另外通过添加类型标示，明确指出函数将文本转换成了由文本值组成的列表。如果没有类型标示，将会有两个不同的 split()潜在实现：分割字节数组的和分割字符串的。在 Python 3 中，类型 Text 是 str 的别名。

函数的返回结果是由行数据组成的可迭代序列，每行包含 3 个字符串，分别代表路径上一个点的经度、纬度和海拔高度。至此，数据仍不可用，需要进一步抽取经度值和纬度值，并把它们转换为浮点数。

底层解析方法将原始数据转换为可迭代元组（或者序列），使得我们能以相对简单一致的方法处理数据文件。第 3 章介绍了如何将 CSV（comma separated values，逗号分隔值）文件转换为元组序列。第 6 章将详述该话题，介绍不同的解析方法。

上面函数的解析结果如下所示：

```
[['-76.33029518659048', '37.54901619777347', '0'],
 ['-76.27383399999999', '37.840832', '0'],
 ['-76.459503', '38.331501', '0'],
 etc.
 ['-76.47350299999999', '38.976334', '0']]
```

每一行是<ns0:coordinates>标记文本经其中的逗号分隔之后得到的列表，包括东西向经度、北南向纬度及海拔高度。稍后将继续创建函数来处理这些计算结果，以得到最终可用的数据。

4.2.2 使用高级方法解析文件

完成了底层解析语法后，就可以将原始数据重构为 Python 程序可用的形式。这类结构化方法适用于 XML、JSON、CSV 以及其他多种序列化数据的物理格式。

我们的目标是编写一组生成器函数，将数据解析为应用可用的形式。生成器函数对 row_iter_kml()函数搜索到的文本执行一些简单的转换，如下所示：

- 舍弃海拔高度属性，保留经度和纬度属性；
- 改变顺序，将(longitude, latitude)改为(latitude, longitude)。

下面的辅助函数可以将这两次转换表示为一致的语法形式：

```
def pick_lat_lon(
  lon: Text, lat: Text, alt: Text) -> Tuple[Text, Text]:
    return lat, lon
```

这里创建了一个接收三个参数的函数，基于其中两个创建了二元组。其中的类型标示比函数本身还复杂。

如下所示使用该函数：

```
from typing import Text, List, Iterable

Rows = Iterable[List[Text]]
LL_Text = Tuple[Text, Text]
def lat_lon_kml(row_iter: Rows) -> Iterable[LL_Text]:
    return (pick_lat_lon(*row) for row in row_iter)
```

该函数将 `pick_lat_lon()` 函数应用于源数据的每一行，`*row` 参数负责将每个三元组转换为三个独立的值作为 `pick_lat_lon()` 函数的参数，这个函数再从每个三元组中抽取数据并重新排序。

为了简化函数定义，我们创建了两个类型别名：`Rows` 和 `LL_Text`。类型别名既可以简化函数定义，又可以复用于其他函数，确保它们能处理相同类型的对象。

这种函数设计模式允许后续替换其中任何函数，以此简化代码重构。可以通过为不同的函数提供多种实现方法来达到这个目标。原则上，高质量的函数式语言编译器也会在优化过程中执行替换操作。

接下来组合上述函数，解析数据文件，构建可用的数据结构，如以下代码所示：

```
url = "file:./Winter%202012-2013.kml"
with urllib.request.urlopen(url) as source:
    v1= tuple(lat_lon_kml(row_iter_kml(source)))
print(v1)
```

其中用 `urllib` 命令打开的数据源是一个本地文件，也可以用它打开远端服务器上的 KML 文件。使用这种文件打开方式旨在确保不论数据源如何，都能以统一的方式打开。

这个脚本基于两个函数实现对 KML 文件的底层解析，`row_iter_kml(source)` 函数生成一个文本序列，`lat_lon_kml()` 函数抽取经度值和纬度值并重新排序。这样就为后续处理提供了中间结果，保证原始数据格式不影响后续处理流程。

运行上述代码，可得到如下结果：

```
(('37.54901619777347', '-76.33029518659048'),
 ('37.840832', '-76.27383399999999'),
 ('38.331501', '-76.459503'),
 ('38.330166', '-76.458504'),
 ('38.976334', '-76.47350299999999'))
```

这样就从复杂的 XML 文件中，采用基本纯函数的方式解析出了经度值和纬度值。由于生成结果是可迭代的，后续可使用函数式编程技术处理从文件中得到的每个点。

至此，就分别实现了底层的 XML 解析函数和上层的数据处理函数。解析 XML 后生成的是一组字符串元组，与 CSV 解析器生成的结果格式兼容，之后从 SQL 数据库中获取数据时，返回的结果也是类似的，这样就保证了上层处理函数能以相同的方法解析不同来源的数据。

下面将通过一系列方法把一组字符串转换为沿着一条路径的一系列数据点。该过程包含多次转换，既要重组数据，还要把字符串转换为浮点数。我们还会介绍简化和明晰后续处理步骤的方法，后续章节会继续使用这些数据，因为它们特别复杂。

4

4.2.3　组对序列元素

将点序列重组为“开始–结束”对组成的序列，是常用的数据处理步骤：根据已有序列 $S=\{s_0,s_1,s_2,\cdots,s_n\}$，生成对序列 $\hat{S}=\{(s_0,s_1),(s_1,s_2),(s_2,s_3),\cdots,(s_{n-1},s_n)\}$。源数据中的第一个元素和第二个元素组成第一对，第二个元素和第三个元素组成第二对，以此类推。进行时域分析时，往往要合并间隔很大的数据，但这里合并相邻的数值就行了。

有了对序列，就可以利用 haversine 函数计算出每对中两点间的距离，在地理位置应用中常用这项技术把一条路径上的点转换为一系列线段。

为什么将序列元素组对，而不做如下处理：

```
begin = next(iterable)
for end in iterable:
    compute_something(begin, end)
    begin = end
```

显然，这样也可以将每段数据作为一个“开始–结束”对来处理，但这种方式将处理函数和重组数据的循环紧密绑定在一起了，使得代码难以复用。由于与 compute_something() 函数绑定在一起，算法很难单独测试组。

这种组合方式也使得开发者难以修改应用的配置，无法简单地用其他实现代替 compute_something() 函数。另外，显式状态和 begin 变量的存在使得未来可能的重构复杂化了。当我们想在循环体中增加新特性时，如果去掉其中某个点，很可能导致 begin 变量设置错误，包含 if 语句的 filter() 函数也会导致 begin 变量更新出现故障。

为了保证代码可复用，需要把这个简单的组对函数独立出来。长远来看，这符合设计目标：把有用的基础函数（例如这里的组对函数）放在单独的库里，便于我们更快、更有把握地解决遇到的问题。

有很多种方法可以把一条路径上的所有点处理成“开始–结束”形式，下面介绍其中几种方

法，第 5 章和第 8 章会基于纯函数风格以递归方式实现组对。

沿着一条路径依次将各个点组对的一个版本如下：

```
from typing import Iterator, Any
Item_Iter = Iterator[Any]
Pairs_Iter = Iterator[Tuple[float, float]]
def pairs(iterator: Item_Iter) -> Pairs_Iter:
    def pair_from(
            head: Any,
            iterable_tail: Item_Iter) -> Pairs_Iter:
        nxt= next(iterable_tail)
        yield head, nxt
        yield from pair_from(nxt, iterable_tail)

    try:
        return pair_from(next(iterator), iterator)
    except StopIteration:
        return iter([])
```

其核心部分是内部函数 `pair_from()`，它的参数是可迭代对象的第一个元素和它自身。首先从第二个参数，即可迭代对象中拿出头元素，与第一个参数组成一对作为结果返回，再递归调用自身，返回剩余的数据对。

类型标示要求参数 `iterator` 的类型是 `Item_Iter`，返回结果的类型是 `Pairs_Iter`，即元素类型为 `float` 的二元组组成的迭代器。mypy 工具通过检查类型标示确保代码可以正常运行。类型标示是在 `typing` 模块中定义的。

输入必须是实现了 `next()`函数的迭代器，为了处理集合数据，需要使用 `iter()`函数显式地将集合转换为迭代器。

`pairs()`函数内调用了 `pair_from()`函数，`pairs()`负责从参数中得到初始值。当输入参数是空迭代器时，`next()`函数的第一次调用将生成一个 `StopIteration` 异常，然后返回一个空的可迭代对象。



Python 的迭代递归使用 `for` 循环处理数据并从递归中返回结果。如果使用看上去更简单的 `return pair_from(nxt, iterable_tail)`方法，你会发现它不能正确地处理可迭代对象并返回所有值。生成器函数中的递归需要使用 `yield` 从可迭代对象中获取结果，这里使用的是 `yield from recursive_iter(args)`。如果改用 `return recursive_iter(args)`，将会返回生成器对象，而不是对函数求值并返回生成的结果。

如果希望使用尾调用优化，可以用生成器表达式代替递归，用 `for` 循环优化递归，下面的函数是沿着路径依次组对的另一个版本：

```
from typing import Iterator, Any, Iterable, TypeVar
T_ = TypeVar('T_')
Pairs_Iter = Iterator[Tuple[T_, T_]]
def legs(lat_lon_iter: Iterator[T_]) -> Pairs_Iter:
    begin = next(lat_lon_iter)
    for end in lat_lon_iter:
        yield begin, end
        begin = end
```

这个版本的实现处理速度非常快，且不受栈长度限制。它可以处理任何类型的序列，将序列生成器给出的值组对。由于循环体内部没有处理函数，需要时即可复用 `legs()`函数。

通过 `TypeVar` 函数定义的类型变量 `T_`用于准确描述 `legs()`函数重组数据的方式。类型标示指出输入类型决定了输出类型。输入类型是某种类型 `T_`组成的迭代器，与输出元组的元素类型必须一致，并且不涉及其他转换。

`begin` 和 `end` 变量保存计算状态，使用有状态变量不符合函数式编程尽量使用不可变数据的要求，因此需要进一步优化。此外，它对函数的使用者是不可见的，是一种 Python 式混合实现风格。

该函数能生成如下对序列：

```
list[0:1], list[1:2], list[2:3], ..., list[-2:]
```

该函数的功能也可表示如下：

```
zip(list, list[1:])
```

最后这种表达方式虽然很直观，但只能用于序列对象，而 `legs()`和 `pairs()`函数适用于任何可迭代对象，包括序列对象。

4.2.4　显式使用 `iter()`函数

在纯粹的函数式语境中，递归函数处理所有可迭代对象，状态只存在于递归调用栈中。在实践中，Python 的可迭代对象往往涉及对其他 `for` 循环的求值。有两种常见的场景：集合和可迭代对象。当用于集合时，由 `for` 语句创建迭代器对象；当用于生成器函数时，生成器函数本身就是迭代器，自己维护内部状态。在 Python 编程的大多数实践中，二者作用相同，但在某些特殊情况下，例如需要显式使用 `next()`函数时，二者的用法并不相同。

前面的 `legs()`函数使用了显式的 `next()`方法从可迭代对象中取值。对于可迭代对象，包括生成器函数和生成器表达式，使用这个函数完全没有问题，但对于序列对象，例如元组和列表，不能用这个函数。

下面 3 个例子说明了 `next()`函数和 `iter()`函数的作用：

```
>>> list(legs(x for x in range(3)))
[(0, 1), (1, 2)]
>>> list(legs([0,1,2]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in legs
TypeError: 'list' object is not an iterator
>>> list(legs(iter([0,1,2])))
[(0, 1), (1, 2)]
```

第一个例子中，legs()函数的参数是可迭代对象，也是生成器表达式，这种情况符合设计目标，数据正确组对，将三个位置点转换为两个路径段。

第二个例子尝试将序列用作 legs()函数的参数，结果发生了错误。虽然在 for 语句中列表对象和可迭代对象的用法是一样的，但这并不适用于所有场景。序列不是迭代器，不能用作 next()函数的参数，for 语句为了处理这两类情形，自动基于序列创建了一个迭代器。

对于第二种情形，需要显式地基于列表对象创建一个迭代器，这样 legs()函数就可以通过迭代器获取列表中的第一个值了。iter()函数负责将列表转换为迭代器。

4.2.5　扩展简单循环

对简单循环的扩展包括两种情况。首先介绍过滤式扩展，这种扩展能够按照某种标准舍弃数据源中的某些数据，这些数据可能是异常值，或者存在格式错误。第二种情况是简单转换原始数据，在原有对象的基础上创建新的对象。这里是要将字符串转换为浮点数。不过如何通过映射扩展简单循环要看具体情况。下面介绍重构 pairs()函数的方法，如果想调整点的顺序并舍弃某些值，应该怎么做呢？可以使用过滤式扩展筛选数据。

在之前循环体的实现中，为了将复杂度降到最低，仅仅返回了数据对，没有其他任何应用逻辑相关的处理。实现的简洁避免了开发者在复杂的状态变换中迷失方向。

为循环添加过滤式扩展的一种实现方案如下：

```
from typing import Iterator, Any, Iterable
Pairs_Iter = Iterator[Tuple[float, float]]
LL_Iter = Iterable[
    Tuple[Tuple[float, float], Tuple[float, float]]]
def legs_filter(lat_lon_iter: Pairs_Iter) -> LL_Iter:
    begin = next(lat_lon_iter)
    for end in lat_lon_iter:
        if #some rule for rejecting:
            continue
        yield begin, end
        begin = end
```

这样在循环体中添加了筛选规则来滤掉某些值，在保持循环简洁明了的前提下，处理过程能够按照预期进行。另外，由于该函数可以处理任何可迭代对象，而不必考虑数据对的具体用途，

为它编写测试用例也很简单。

这里没有实现#some rule for rejecting 部分的代码，这部分用于通过 begin、end 或者二者结合过滤掉某些无效数据。例如需要过滤掉 begin == end，以免出现长度为 0 的线段。

接下来的重构工作是为循环添加映射。在应用功能、设计目标不断变化的过程中，经常需要加入映射。这里的原始数据是字符串类型的，需要将它们转换为浮点数以备后用。这个转换本身不复杂，展示了这种扩展的实现方法。

下面的示例把生成器函数包裹在生成器表达式中，实现了数据映射。

```
trip = list(
    legs(
        (float(lat), float(lon))
        for lat,lon in lat_lon_kml(row_iter_kml(source))
    )
)
```

legs()函数的输入参数是一个生成器表达式，该生成器表达式把使用 lat_lon_kml()的输出转换为浮点数。或者反过来说：首先把 lat_lon_kml()函数的输出转换为浮点数对，再转换为一系列路径段。

现在事情变得有点复杂了，几个函数层层嵌套，在数据生成器上分别应用了 float()、legs()以及 tuple()。对于这类复杂的表达式，常用的重构方法是把生成器表达式从实例化的集合中独立出来，对上述表达式的一种简化实现如下：

```
flt = (
    (float(lat), float(lon))
    for lat,lon in lat_lon_kml(row_iter_kml(source))
)
print(tuple(legs(ll_iter)))
```

这里将生成器函数赋给了变量 flt，该变量不是集合对象，也不是通过列表推导生成的对象，只是给生成器表达式起了个名字（flt），然后在其他表达式中使用这个名字。

对 tuple()的求值导致作为参数的惰性变量开始求值，flt 变量对应的对象只在需要求值的时候才被创建。

除了上面的代码实现，还有其他方法可以实现重构。数据源经常发生变化。在上面的例子中，lat_lon_kml()函数与其他表达式紧密绑定，导致处理其他数据源时，难以复用这个函数。

当需要把上面的转换过程 float()参数化，以便之后复用，可以基于生成器表达式专门定义一个函数。下面抽取一些处理过程放在单独的函数里，来对操作过程进行分组。这里由于字符串到浮点数的转换与数据源无关，可以把这个复杂的转换表达式写入下面的函数中。

```
from typing import Iterator, Tuple, Text, Iterable
Text_Iter = Iterable[Tuple[Text, Text]]
LL_Iter = Iterable[Tuple[float, float]]
def float_from_pair(lat_lon_iter: Text_Iter) -> LL_Iter:
    return (
        (float(lat), float(lon))
        for lat,lon in lat_lon_iter
    )
```

该函数对可迭代对象每组值的第 1 个数据和第 2 个数据应用 `float()` 函数，把原始数据转换成由浮点数组成的二元组，这里借助了 Python 的 `for` 语句解析此二元组。

类型标示要求输入是 `Text_Iter` 类型：`Text` 类型二元组组成的可迭代对象；返回值是 `LL_Iter` 类型：浮点数二元组组成的可迭代对象。`LL_Iter` 类型可以用于其他复杂函数的定义中。

该函数可用于如下场景：

```
legs(
    float_from_pair(
        lat_lon_kml(
            row_iter_kml(source))))
```

这样就把 KML 文件中的数据转换为浮点数了。由于每一个处理环节都是一个简单的前缀式函数，可以方便地可视化整个过程，每个函数的输入都是内层嵌套函数的输出。

解析过程的输入值通常是字符串，对于数据处理应用来说，经常需要将输入值转换为浮点数、整数、十进制数等类型，这时就要在清洗源数据的表达式中插入类似于 `float_from_pair()` 的函数。

原先字符串类型的输出如下所示：

```
(('37.54901619777347', '-76.33029518659048'),
 ('37.840832', '-76.27383399999999'),
 ...
 ('38.976334', '-76.47350299999999'))
```

所需的浮点型数据如下所示：

```
(((37.54901619777347, -76.33029518659048),
 (37.840832, -76.273834)),
 ...
 ((38.330166, -76.458504), (38.976334, -76.473503)))
```

接下来简化转换流程，前面把代码优化成了 `flt = ((float(lat), float(lon)) for lat, lon in lat_lon_kml())`，根据函数替换规则，可以把类似于 `((float(lat), float (lon)) for lat,lon in lat_lon_kml())` 的复杂表达式，替换为返回值相同的函数，这里是 `float_from_pair(lat_lon_kml())`。这样的重构可以简化复杂的表达式，同时保证功能不变。

第 5 章将继续讨论代码简化问题，第 6 章会介绍如何把简化后的函数应用于文件解析过程中。

4.2.6 将生成器表达式应用于标量函数

下面介绍如何使用复杂的生成器表达式转换数据类型，以及如何把复杂函数应用于生成器创建的单个值。

这些非生成器函数是“标量的”，因为它们处理的是单个标量值。为了处理集合数据，需要把标量函数嵌入生成器表达式中。

继续前面的例子，首先创建 haversine 函数，然后使用生成器表达式将标量的 haversine 函数应用于从 KML 文件中提取的数值。

haversine 函数的具体实现如下：

```
from math import radians, sin, cos, sqrt, asin
from typing import Tuple
MI = 3959
NM = 3440
KM = 6371
Point = Tuple[float, float]
def haversine(p1: Point, p2: Point, R: float=NM) -> float:
    lat_1, lon_1 = p1
    lat_2, lon_2 = p2
    Δ_lat = radians(lat_2 - lat_1)
    Δ_lon = radians(lon_2 - lon_1)
    lat_1 = radians(lat_1)
    lat_2 = radians(lat_2)
    a = sqrt(
        sin(Δ_lat / 2) ** 2 +
        cos(lat_1) * cos(lat_2) * sin(Δ_lon / 2) ** 2
    )
    c = 2 * asin(a)
    return R * c
```

这个实现比较简单，在网上搜索一下，找到代码并复制过来即可。这里给起点、终点和返回值添加了类型标示，显式类型声明 Point = Tuple[float, float]使得 mypy 工具可以正确地使用函数。

下面的代码演示了如何使用前面定义好的函数处理 KML 文件中的数据，并计算出距离序列：

```
trip = (
    (start, end, round(haversine(start, end), 4))
    for start, end in
        legs(float_from_pair(lat_lon_kml()))
)

for start, end, dist in trip:
    print(start, end, dist)
```

关键部分是赋给 trip 变量的生成器表达式。我们用起点、终点以及两点间的距离组成三元

组，其中起点和终点来自 legs()函数，用它来处理从 KML 文件中提取的经纬度数值对转换而来的浮点数据。

计算结果如下所示：

```
(37.54901619777347, -76.33029518659048) (37.840832, -76.273834) 17.7246
(37.840832, -76.273834) (38.331501, -76.459503) 30.7382
(38.331501, -76.459503) (38.845501, -76.537331) 31.0756
(36.843334, -76.298668) (37.549, -76.331169) 42.3962
(37.549, -76.331169) (38.330166, -76.458504) 47.2866
(38.330166, -76.458504) (38.976334, -76.473503) 38.8019
```

这样就简洁地定义了每个处理步骤。整个处理过程由函数和生成器表达式组成，总体而言也很简洁。

当然，这里得到的输出数据还需要进一步处理，比如首先需要用字符串的 format()方法把输出数据整理得易读一些。

更重要的是，需要从数据中提取一些累积值，即对已有数据进行归约。比如可以提取数据的最大纬度值和最小纬度值，从而得到一条路线的最南点和最北点。或者通过归约数据，找到路径段中最长的距离，或者所有距离的和。

使用 Python 处理数据时有个问题：trip 变量只能使用一次，无法对其进行多次归约，而 itertools.tee()函数能够多次使用可迭代对象，毕竟每次归约都要重新读取并解析 KML 文件，成本太高了。

将中间计算结果实例化可以提高计算效率，稍后详述并介绍如何对已有数据进行多次归约。

4.2.7　用 any()函数和 all()函数进行归约

可以使用函数 any()和 all()进行布尔归约，将一个集合的元素归约成单个值（True 或者 False）。all()函数确保所有值都是 True，any()函数确保至少有一个值是 True。

这两个函数与数理逻辑中的全称量词与存在量词密切相关，当我们要表达某个给定集合中的所有元素具备某一属性时，可以写成下面的形式：

$$(\forall_{x \in \text{Someset}})\ \text{Prime}(x)$$

这个公式读作：对于 SomeSet 集合中的任一元素 x，函数 Prime(x)为真。注意在逻辑表达式前面添加的量词。

使用 Python，需要稍微调整逻辑表达式中各项的顺序，如下所示：

```
all(isprime(x) for x in SomeSet)
```

对每个输入参数（isprime(x)）求值，最终将集合归约为单个值 True 或者 False。

any()函数与存在量词相关，当我们想确认集合中没有元素是质数时，可以使用下面任一表达式：

$$\neg(\forall_{x \in \text{SomeSet}})\text{Prime}(x) \equiv (\exists_{x \in s})\neg\text{Prime}(x)$$

第一个公式表示：SomeSet 集合中的元素不都是质数。第二个公式表示：SomeSet 集合中至少有一个元素不是质数。这两种表述效果相同：如果不是所有元素都是质数，那么至少有一个元素是非质数。

略微调整一下顺序，上面公式的 Python 版本如下：

```
not all(isprime(x) for x in someset)
any(not isprime(x) for x in someset)
```

二者作用相同，差别在于性能和可读性。又由于二者的性能差别不大，就只能通过可读性来区分了，那么上面哪种表述方式更易读呢？

all()函数对集合中的所有元素进行 and 归约，效果相当于在各个值之间加上 and 运算符。类似地，any()函数进行 or 归约。第 10 章将继续介绍 reduce()函数。这里没有最佳答案，不同的读者对可读性各有偏好。

下面看一下这些函数的退化形式，如果作为输入参数的序列的元素数量为 0 会怎样？all(()) 或者 all([])的值会是什么？

如果我们问："空集合中的元素都是质数吗？"你会怎么回答呢？作为提示，我们稍微拓展一下，看看单位元的概念。

如果我们问："空集中所有元素都是质数，并且 SomeSet 集合中所有元素都是质数吗？"可以把对空集的归约和对 SomeSet 的归约放在一起做交集来解决这个问题。

$$(\forall_{x \in \varnothing})\text{Prime}(x) \land (\forall_{x \in \text{SomeSet}})\text{Prime}(x)$$

使用集合的分配律处理上面的 and 运算符，改为两个集合做并集的形式，再测试元素是否为质数。

$$(\forall_{x \in \varnothing \cup \text{SomeSet}})\text{Prime}(x)$$

显然，$\text{SomeSet} \cup \varnothing \equiv \text{SomeSet}$，任意集合与空集的并集是这个集合自身，所以空集是**并集单位元**，与 0 作为加法单位元是一个道理。

$$a + 0 = a$$

类似地，any(())一定是 or 单位元，即 False。由于 $b \times 1 = b$，可以把 1 看作乘法单位元，

所以 all(())一定是 True。

验证一下 Python 的行为是否符合如下规则：

```
>>> all(())
True
>>> any(())
False
```

可以用 Python 的一些工具处理逻辑运算，包括内置的 and、or、not 运算符，及用于集合的 any()、all()函数等。

4.2.8　使用 **len()**和 **sum()**

len()和 sum()提供了两种简单的归约方法：计算序列中所有值的个数和汇总值。这两个函数在数学上相近，但在 Python 中的实现方法却有很大差别。

从数学角度看，这两个函数的高度相似性体现在：len()函数把序列中每个元素看作 1，然后返回汇总值：$X:\sum_{x\in X}1=\sum_{x\in X}x^0$；sum()函数则取序列中每个元素的实际值，然后返回汇总值：$X:\sum_{x\in X}x=\sum_{x\in X}x^1$。

sum()函数可用于任何可迭代对象，len()函数不能用于可迭代对象，只能用于序列。这种实现方法上的不对等导致在某些情况下，开发统计算法会遇到一些小麻烦。

对于空序列，两个函数都返回加法单位元 0。

```
>>> sum(())
0
```

虽然 sum(())返回整数 0，但在计算其他类型的数值时，整数 0 将被强制转换为与输入数据匹配的类型。

4.2.9　使用汇总和计数进行统计分析

基于函数 sum()和 len()，可以给出算术平均值的简单定义，如下所示：

```
def mean(items):
    return sum(items) / len(items)
```

虽然这个定义很简洁，却不能用于可迭代对象，只能用于支持 len()函数的集合，这一点通过添加类型标示很容易发现。定义 mean(items: Iterable) -> float 不成立，因为 Iterable 类型不支持 len()函数。

诚然，对可迭代对象即使是求简单的平均值或者标准差都是很困难的。在 Python 中，我们要么实例化这些序列对象，要么转而使用一些更复杂的运算来完成求值。

正确的定义方法如下：

```
from collections import Sequence
def mean(items: Sequence) -> float:
    return sum(items) / len(items)
```

定义正确的类型标示以确保 `sum()`和 `len()`能正常工作。

下面的定义用简洁的表达式定义了平均值和标准差。

```
import math
s0 = len(data)  # sum(x ** 0 for x in data)
s1 = sum(data)  # sum(x ** 1 for x in data)
s2 = sum(x ** 2 for x in data)
mean = s1 / s0
stdev = math.sqrt(s2 / s0 - (s1 / s0) ** 2)
```

`s0`、`s1` 和 `s2` 是 3 种求和，实现过程都比较简单，结构也类似。通过它们可以很方便地算出平均值。计算标准差虽然稍复杂一些，但仍然可行。

更复杂的统计函数仍然具备这种良好的对称性，包括相关度和最小方差线性回归。

两个样本之间的相关度可以通过它们的标准值计算出来，如下所示：

```
def z(x, μ_x: float, σ_x: float) -> float:
    return (x - μ_x) / σ_x
```

计算过程是对每个样本 `x` 减去平均值 `μ_x`，再除以标准差 `σ_x`，返回结果以 `σ` 为单位。大约 2/3 的情况下偏差在±1σ 范围内，偏差越大，出现的概率越小，超过±3σ 的概率则小于 1/100。

如下所示使用该标量函数：

```
>>> d = [2, 4, 4, 4, 5, 5, 7, 9]
>>> list(z(x, mean(d), stdev(d)) for x in d)
[-1.5, -0.5, -0.5, -0.5, 0.0, 0.0, 1.0, 2.0]
```

这里首先对变量 `d` 中的值进行归一化，把计算结果实例化到一个列表中。使用生成器表达式，将标量函数 `z()`应用于序列对象。

利用上面的函数，可以给出函数 `mean()`和 `stdev()`的实现方法。

```
def mean(samples: Sequence) -> float:
    return s1(samples) / s0(samples)
def stdev(samples: Sequence) -> float:
    N = s0(samples)
    return sqrt((s2(samples) / N) - (s1(samples) / N) ** 2)
```

用类似的方法，如下改写前面 3 个求和函数：

```
def s0(samples: Sequence) -> float:
    return sum(1 for x in samples) # or len(data)
```

```
def s1(samples: Sequence) -> float:
    return sum(x for x in samples) # or sum(data)
def s2(samples: Sequence) -> float:
    return sum(x * x for x in samples)
```

尽管简洁明了，但该实现仍无法应用于可迭代对象。计算平均值时，需要对可迭代对象做一次汇总和一次计数。计算标准差时，需要做两次汇总和一次计数，针对这类统计相关的处理，必须将序列对象实例化，才能多次使用数据。

计算两个样本集相关性的函数实现如下：

```
def corr(samples1: Sequence, samples2: Sequence) -> float:
    m_1, s_1 = mean(samples1), stdev(samples1)
    m_2, s_2 = mean(samples2), stdev(samples2)
    z_1 = (z( x, m_1, s_1 ) for x in samples1)
    z_2 = (z( x, m_2, s_2 ) for x in samples2)
    r = (sum(zx1 * zx2 for zx1, zx2 in zip(z_1, z_2))
        / len(samples1))
    return r
```

上面的相关性计算用到了样本集的基本统计特征值：平均值和标准差。基于这些特征值，我们定义了两个生成器函数，计算每个样本集的归一值。接着用 `zip()` 函数（见下个示例）把两个归一化序列中的元素组对，计算每对的乘积。这个乘积序列的平均值即表示两个样本集的相关度。

计算两个样本集之间的相关度如下所示：

```
>>> # Height (m)
>>> xi = [1.47, 1.50, 1.52, 1.55, 1.57, 1.60, 1.63, 1.65,
...     1.68, 1.70, 1.73, 1.75, 1.78, 1.80, 1.83,]
>>> # Mass (kg)
>>> yi = [52.21,53.12,54.48,55.84,57.20,58.57,59.93,61.29,
...     63.11, 64.47, 66.28, 68.10, 69.92, 72.19, 74.46,]
>>> round(corr( xi, yi ), 5)
0.99458
```

这里的两个数据点序列 `xi` 和 `yi` 的相关度超过了 0.99，表明二者关系紧密。

以上代码示例很好地体现了函数式编程的优点。我们用几个函数构建出了一个简单易用的统计模块，且每个函数只是简单的表达式。作为反例，不妨假设把 `corr()` 函数写成一个长而复杂的表达式，函数内部有很多只用了一次的内部变量，用复制粘贴的方法把它们替换成各自代表的表达式。不难看出，虽然这里的 `corr()` 函数仅由 6 行 Python 代码组成，但仍属于函数式编程。

4.3　使用 zip() 函数实现结构化和平铺序列

`zip()` 函数将来自多个可迭代对象或者序列的数据交叉组合在一起，将带有 n 个元素的可迭代对象或者序列转换为 n 元组。前面的例子用 `zip()` 函数将两个样本集的数据组合在一起，生成了一个二元组序列。



zip()函数返回一个生成器，而不会将返回结果实例化。

zip()函数的作用如下：

```
>>> xi = [1.47, 1.50, 1.52, 1.55, 1.57, 1.60, 1.63, 1.65,
... 1.68, 1.70, 1.73, 1.75, 1.78, 1.80, 1.83,]
>>> yi = [52.21, 53.12, 54.48, 55.84, 57.20, 58.57, 59.93, 61.29,
... 63.11, 64.47, 66.28, 68.10, 69.92, 72.19, 74.46,]
>>> zip(xi, yi)
<zip object at 0x101d62ab8>
>>> list(zip(xi, yi))
[(1.47, 52.21), (1.5, 53.12), (1.52, 54.48), (1.55, 55.84),
 (1.57, 57.2), (1.6, 58.57), (1.63, 59.93), (1.65, 61.29),
 (1.68, 63.11), (1.7, 64.47), (1.73, 66.28), (1.75, 68.1),
 (1.78, 69.92), (1.8, 72.19), (1.83, 74.46)]
```

zip()函数需要处理一些特殊情况，比如我们需要知道下列情形中它的运行方式。

- 如果没有输入参数会怎样？
- 如果只有一个输入参数会怎样？
- 如果作为输入参数的序列长度不一致会怎样？

对于其他函数（例如 any()、all()、len()和 sum()等），我们需要知道归约空序列时的单位元是什么。例如 sum(())返回 0，借助这个概念可以求得 zip()函数的单位元。

显然，每种特殊情况都会生成某种可迭代对象，下面的代码演示了这些情况下 zip()函数的行为。空输入参数的情况如下：

```
>>> zip()
<zip object at 0x101d62ab8>
>>> list(_)
[]
```

可见没有输入参数的 zip()函数返回一个生成器函数，但里面不含任何数据项，这符合输出为可迭代对象的要求。

单个输入参数的情况如下：

```
>>> zip((1,2,3))
<zip object at 0x101d62ab8>
>>> list(_)
[(1,), (2,), (3,)]
```

这种情况下，zip()函数返回单元组序列，也是符合要求的。

输入序列的长度不同时 zip()函数的行为如下：

```
>>> list(zip((1, 2, 3), ('a', 'b')))
[(1, 'a'), (2, 'b')]
```

对于这个结果，Python 社区内部存在争议，为什么要截断？为什么不用 None 值填充较短的列表？作为 zip()函数的替代，itertools 模块的 zip_longest()函数满足上面的要求，第 8 章会介绍它。

4.3.1　将压缩序列解压

zip()映射是可以逆操作的，也就是解压，下面介绍几种解压元组集合的方法。



当数据被多次处理后，就无法将可迭代对象组成的元组完全解压了。根据具体情况，可能需要将可迭代对象实例化，以提取需要的数据。

之前讲过第一种方法了：使用生成器函数解压元组序列。例如，假设下面的 pairs 是由二元组组成的序列：

```
p0 = (x[0] for x in pairs)
p1 = (x[1] for x in pairs)
```

这样就可以得到两个序列：p0 序列由二元组序列的第一个元组组成，p1 序列由二元组序列的第二个元素组成。

某些情况下，需要使用 for 循环提供的多重赋值方法来拆解元组，例如计算乘积之和的方法如下：

```
sum(p0 * p1 for for p0, p1 in pairs)
```

这里用 for 语句把每个二元组拆解到了 p0 和 p1 中。

4.3.2　平铺序列

有时压缩后的数据需要平铺成一维序列，例如输入文件的结构可能如下所示：

```
2     3     5     7     11    13    17    19    23    29
31    37    41    43    47    53    59    61    67    71
...
```

可以用(line.split() for line in file)生成一个序列，序列的每个元素是文件中一行数据构成的十元组。

得到的数据结构如下所示：

```
>>> blocked = list(line.split() for line in file)
>>> blocked
[['2', '3', '5', '7', '11', '13', '17', '19', '23', '29'], ['31', '37',
'41', '43', '47', '53', '59', '61', '67', '71'], ['179', '181', '191',
'193', '197', '199', '211', '223', '227', '229']]
```

但这并不是我们想要的最终结果，我们希望得到的结构是一维序列，但实际输入的每个元素都是十元组。每次拆解一个这样的元组则太过麻烦。

平铺这类数据结构，可以使用如下所示的双层生成器表达式：

```
>>> (x for line in blocked for x in line)
<generator object <genexpr> at 0x101cead70>
>>> list(_)
['2', '3', '5', '7', '11', '13', '17', '19', '23', '29', '31',
 '37', '41', '43', '47', '53', '59', '61', '67', '71',
 ... ]
```


第一个 for 从句从 blocked 列表中解析出对象，每个对象是一个长度为 10 的列表，赋给 line 变量，第二个 for 从句从 line 变量中解析出字符串，赋给 x 变量，包含最终结果的生成器保存在 x 变量中。

改写成如下形式更易于理解：

```
def flatten(data: Iterable[Iterable[Any]]) -> Iterable[Any]:
    for line in data:
        for x in line:
            yield x
```

改写后可以看到生成器表达式的工作方式：外层 for 从句（for line in data）遍历输入数据中的每个十元组，内层 for 从句（for x in line）遍历外层语句中的每个元素。

这样双层序列复合型结构就转换成了单层序列。它能够将任何嵌套的双层可迭代对象转换为单层可迭代对象，例如双层列表、列表–集合组合结构等。

4.3.3　结构化一维序列

有时需要通过某种形式将一维序列转换为复合序列，与前面的处理过程相比，可能更麻烦一些。可以用 itertools 模块的 groupby()函数进行处理，第 8 章会详述。

假设现有如下一个一维列表：

```
flat = ['2', '3', '5', '7', '11', '13', '17', '19', '23', '29',
 '31', '37', '41', '43', '47', '53', '59', '61', '67', '71',
 ... ]
```

使用嵌套生成器函数，可以把它从一维序列转换为复合序列。为了实现这一点，需要一个可多次使用的简单迭代器，表达式如下所示：

```
>>> flat_iter = iter(flat)
>>> (tuple(next(flat_iter) for i in range(5))
...     for row in range(len(flat)//5)
... )
<generator object <genexpr> at 0x101cead70>
```

```
>>> list(_)
[('2', '3', '5', '7', '11'),
 ('13', '17', '19', '23', '29'),
 ('31', '37', '41', '43', '47'),
 ('53', '59', '61', '67', '71'),
 ...
]
```

首先，在双层循环外部创建了一个用于生成复合序列的迭代器，生成器表达式使用 `tuple(next(flat_iter) for i in range(5))`，基于变量 `flat_iter` 中的可迭代对象创建五元组。该表达式嵌于外层更大的生成器内，此生成器负责按照指定次数循环执行内层循环，生成最终的复合序列。

该表达式只适用于作为输入的一维列表能被子序列平分的情况，如果最后一行有剩余的元素，需要单独处理。

可以先使用上面的函数获得相同长度的元组，然后用下面的函数处理剩余长度不同的部分：

```
ItemType = TypeVar("ItemType")
Flat = Sequence[ItemType]
Grouped = List[Tuple[ItemType, ...]]

def group_by_seq(n: int, sequence: Flat) -> Grouped:
    flat_iter=iter(sequence)
    full_sized_items = list(tuple(next(flat_iter)
        for i in range(n))
            for row in range(len(sequence)//n))
    trailer = tuple(flat_iter)
    if trailer:
        return full_sized_items + [trailer]
    else:
        return full_sized_items
```

首先将长度为 `n` 的元组组成的列表提取到 `full_sized_items` 中，如果元素还有剩余，则将剩余元素组成一个元组，追加到 `full_sized_items` 后面。如果元组长度为 0，则忽略 `trailer` 元组，返回原来的列表。

类型标示中包含了一个类型变量的抽象定义 `ItemType`，用于表示函数的输入类型和输出类型是一致的，字符串序列或者浮点序列皆可。

函数的输入是由数据项组成的序列，输出是一个列表，元素为相同类型的数据项组成的元组。这里的数据项用 `ItemType` 表示，可以是任何类型。

这个实现不如之前的算法简洁，函数式特征也太不明显。可以把它改写成一个简单的生成器函数，生成可迭代对象而不是列表。

下面的代码使用 `while` 循环实现算法逻辑，并结合了尾递归优化：

```
ItemType = TypeVar("ItemType")
Flat_Iter = Iterator[ItemType]
Grouped_Iter = Iterator[Tuple[ItemType, ...]]

def group_by_iter(n: int, iterable: Flat_Iter) -> Grouped_Iter:
    row = tuple(next(iterable) for i in range(n))
    while row:
        yield row
        row = tuple(next(iterable) for i in range(n))
```

该函数从输入可迭代对象中抽取指定长度的行，当迭代至输入数据尾部时，`tuple(next(iterable) for i in range(n))`将会返回一个空元组，这是递归的基础型式，也是 `while` 循环的结束判断条件。

类型标示相应地修改成了包含迭代器的类型,不限于处理序列类型。由于显式使用了 `next()` 函数，需要这样使用：`group_by_iter(7, iter(flat))`。必须使用 `iter()`函数将集合转换为迭代器。

4.3.4 结构化一维序列的另一种方式

假设需要将下列一维列表数据转换为数据对：

```
flat= ['2', '3', '5', '7', '11', '13', '17', '19', '23', '29',
 '31', '37', '41', '43', '47', '53', '59', '61', '67', '71',... ]
```

可以运用列表切片技术：

```
zip(flat[0::2], flat[1::2])
```

切片 `flat[0::2]`表示列表中下标为偶数的所有元素，`flat[1::2]`代表列表中下标为奇数的所有元素。将它们组合成二元组，其中第一个元素来自偶数下标列表，第二个元素来自奇数下标列表。当源列表长度是偶数时，这个方法效果很好。如果长度是奇数，忽略最后一个元素，稍后会给出解决方法。

这个实现的一个优点是简洁。对于相同的问题，前面的实现用的代码更多。

还可以进一步抽象，使用星号参数语法，创建指定长度的复合列表，如下所示：

```
zip(*(flat[i::n] for i in range(n)))
```

函数的返回结果是 `n` 个切片：`flat[0::n]`, `flat[1::n]`, `flat[2::n]`, `...`, `flat[n-1::n]`。这些切片作为 `zip()`函数的参数，将各个元素逐一组合在了一起。

之前讲过 `zip()`函数会按照最短列表进行截断。在上面的例子中，如果列表长度不是分组因素 `n` 的整数倍，即 `len(flat) % n != 0` 时，最后一个分组的长度将小于前面分组的长度，导致前面的分组被截断，这不是我们想要的结果。

如果改用 itertools.zip_longest()方法，最后的分组长度不足的部分会被 None 值补齐，保证所有分组的长度为 n。某些情况下，这些补充值可以接受，而某些情况下我们不希望函数返回这些补充值。

列表切片方法为解决一维列表结构化问题提供了另一种实现方式。作为一种通用实现方法，相比之前的实现，它的优点并不明显。作为将一维列表转换为二元组的实现方法，它的最大优点是简洁。

4.4　使用 reversed()函数改变顺序

有时需要反转序列。Python 提供了两种实现方式：使用 reversed()函数或者使用反序下标值。

例如，假设需要把一个数字转换为 16 进制数或者 2 进制数，下面是一个简单的转换函数：

```
def digits(x: int, b: int) -> Iterator[int]:
    if x == 0: return
    yield x % b
    for d in digits(x // b, b):
        yield d
```

该函数使用递归从最低有效位向最高有效位依次返回计算结果，x % b 的值是以 b 为基底的 x 值的最低有效位。

上述算法可表示如下：

$$\text{digits}(x,b)=\begin{cases}[\,] & x=0\\ [x \bmod b]+\text{digits}\left(\dfrac{x}{b},b\right) & x>0\end{cases}$$

多数情况下，我们希望返回的数值以反序输出。把上面函数的结果包裹在 reversed()函数中，就可以改变输出各位数字的顺序了。

```
def to_base(x: int, b: int) -> Iterator[int]:
    return reversed(tuple(digits(x, b)))
```



reversed()函数返回的是可迭代对象，但它的参数必须是序列。函数将源序列中的元素以反序依次返回。

也可以用类似于 tuple(digits(x, b))[::-1]的方式实现相同的反序效果，但切片不是可迭代对象，而是从实例化对象上变换而来的实例化对象。这里用到的集合较小，所以两种实现方法的区别不大。对于大的数据集合，使用 reversed()函数的实现方法消耗内存更少。

4.5　使用 enumerate()函数包含下标值

Python 的 enumerate()函数可以为序列或者可迭代对象添加下标信息，实际上可以将其看作某种特殊形式的打包，也就 unwrap(process(wrap(data)))设计模式的一部分。

使用方法大致如下所示：

```
>>> xi
[1.47, 1.5, 1.52, 1.55, 1.57, 1.6, 1.63, 1.65, 1.68, 1.7, 1.73,
 1.75, 1.78, 1.8, 1.83]
>>> list(enumerate(xi))
[(0, 1.47), (1, 1.5), (2, 1.52), (3, 1.55), (4, 1.57),
 (5, 1.6), (6, 1.63), (7, 1.65), (8, 1.68), (9, 1.7),
 (10, 1.73), (11, 1.75), (12, 1.78), (13, 1.8), (14, 1.83)]
```

enumerate()函数将输入序列的每一个元素变为一个二元组，其中第一个元素是下标值（序号），另一个是原始输入元素，变换过程大致如下：

```
zip(range(len(source)), source)
```

enumerate()函数的特点是返回结果是可迭代对象，并能以任何可迭代对象为输入值。

在统计处理过程中，可以使用 enumerate()函数在每个样本值前面加序号，将一组简单值转换为一个时间序列。

4

4.6　小结

本章详细介绍了 Python 的几种内置归约方法。

首先使用 any()或者 all()进行关键逻辑处理，实际上相当于用 or 或者 and 运算符进行归约。

接着使用 len()和 sum()归约数值，使用这些函数实现了一些高阶统计处理函数。第 6 章还将继续介绍归约。

然后介绍了 Python 内置的几个映射函数。

zip()函数可用于合并多个序列。借助该函数，可以结构化简单序列，或者将复合序列平铺为一维序列。之后我们将看到，在某些情况下，复合数据易于处理；而在另外一些情况下，一维数据易于处理。

enumerate()函数将可迭代对象映射为二元组序列。每对二元组中，第一个元素是其下标值，第二个元素是输入值本身。

reversed()函数按照相反顺序依次输出输入序列。有些算法适合按照某个顺序给出结果，但需要以相反的顺序呈现结果，该函数正好适合这种情况。

下一章将介绍映射和归约函数如何通过增加函数参数来定制它们的行为。以函数作为参数的函数，是我们最早接触的高阶函数，后面还将介绍以函数为返回值的高阶函数。

第 5 章

高阶函数

函数式编程范式的一个重要特征是高阶函数。高阶函数指以函数为参数，或者以函数为返回值的函数。本章介绍 Python 提供的高阶函数，并进行一定的逻辑拓展。

高阶函数分为以下三类：

- 输入参数中包含一个或多个函数的函数；
- 返回函数的函数；
- 输入参数中包含函数，返回值也是函数的函数，即上面两种情况的结合。

Python 提供了许多第一类的高阶函数，本章主要介绍这类函数。后面的章节会介绍几个提供高阶函数的库模块。

一个函数的返回值是另一个函数，这听上去似乎有点奇怪，以 `Callable` 类的对象为例，当函数的返回值是一个 `Callable` 对象时，实际上就是一个函数返回了另一个函数。

能以函数为参数，或者返回函数的函数还包括复杂的 `Callable` 类以及函数装饰器。本章会涉及函数装饰器，但详述见第 11 章。

前面提到了如果 Python 能提供处理集合函数的高阶版本就好了，本章将介绍 `reduce(extract())` 设计模式，它从大的元组中抽取指定的字段，然后进行归约。本章最后会介绍如何基于这些常见的集合处理函数来自定义高阶函数。

本章会用到下列函数：

- `max()` 和 `min()`
- `map()`
- `filter()`
- `iter()`
- `sorted()`

本章还会介绍用于简化高阶函数定义的匿名函数。

Python 的 itertools 模块中定义了很多高阶函数，第 8 章和第 9 章将详细介绍这些函数。

另外，functools 模块提供了一个通用的 reduce()函数，第 10 章会讲解它。之所以推迟到第 10 章，是因为它的适用场景不如本章的高阶函数广。

函数 max()和 min()是归约函数，其输入是集合，输出是单个值。其他函数映射不会把集合变为单个值。



函数 max()、min()和 sorted()有默认行为模式和高阶行为模式，其函数参数通过可选的 key 参数提供。函数 map()和 filter()则将映射函数作为第一个位置参数。

5.1 用 max()函数和 min()函数寻找极值

max()函数和 min()函数具有双面性，它们可以像普通函数那样应用于集合，也可以用作高阶函数。

其默认行为模式如下：

```
>>> max(1, 2, 3)
3
>>> max((1,2,3,4))
4
```

这两个函数都可以接收无限多个输入参数，也可以将一个序列或者可迭代对象作为单一输入，找到其中的最大（或最小）值。

还可以用它们做一些更复杂的事，以第 4 章中的旅行数据为例，使用函数可以生成如下所示的一系列元组数据：

```
(
 ((37.54901619777347, -76.33029518659048), (37.840832, -76.273834), 17.7246),
 ((37.840832, -76.273834), (38.331501, -76.459503), 30.7382),
 ((38.331501, -76.459503), (38.845501, -76.537331), 31.0756),
 ((36.843334, -76.298668), (37.549, -76.331169), 42.3962),
 ((37.549, -76.331169), (38.330166, -76.458504), 47.2866),
 ((38.330166, -76.458504), (38.976334, -76.473503), 38.8019)
)
```

该集合中的每个元组包含 3 个值：起点、终点和距离，位置数据由经纬度数据组成。东经为正数，所以上面这些数据是美国东海岸的一条路径，大约为西经 76°，距离单位为海里。

可以通过下面三种方法，从这些数据中得到最远距离和最近距离。

- 用生成器函数提取距离值。舍弃每段路径中的其他数据项，只保留距离。如果后续有其他处理流程，用这种方法会造成麻烦。

- ❑ 使用 unwrap(process(wrap()))设计模式，返回包含最长距离和最短距离的路径段。虽然这里只用到了距离信息，但返回结果中包含了关于这段路径的完整信息。
- ❑ 将 max()函数和 min()函数用作高阶函数，定义函数抽取重要距离值。

作为对照，先讲前两种方案。下面的脚本首先构建出旅行数据，然后用前两种方法获得最长距离和最短距离。

```
from ch02_ex3 import (
    float_from_pair, lat_lon_kml, limits, haversine, legs
)
path = float_from_pair(float_lat_lon(row_iter_kml(source)))
trip = tuple(
    (start, end, round(haversine(start, end), 4))
        for start, end in legs(iter(path)))
```

这段脚本中的 source 是一个包含数据点的、打开的 KML 文件对象，trip 对象是包含各个路径段的元组。每个路径段是一个三元组，包含起点、终点和通过 haversine()函数计算出来的距离。legs()函数将源 KML 文件中的路径转换为“起点–终点”对。

得到 trip 对象后，就可以提取距离信息，计算最大值和最小值了，代码如下所示：

```
>>> long = max(dist for start, end, dist in trip)
>>> short = min(dist for start, end, dist in trip)
```

这里使用生成器函数从 trip 元组中提取了需要的数据。由于每个生成器表达式只能用一次，所以生成器表达式需要写两遍。

在一个比前面更大的数据集上运行得到如下结果：

```
>>> long
129.7748
>>> short
0.1731
```

下面用 unwrap(process(wrap()))设计模式实现。清楚起见，把函数命名为 wrap()和 unwrap()，具体实现和运行结果如下：

```
from typing import Iterator, Iterable, Tuple, Any

Wrapped = Tuple[Any, Tuple]
def wrap(leg_iter: Iterable[Tuple]) -> Iterable[Wrapped]:
    return ((leg[2], leg) for leg in leg_iter)
def unwrap(dist_leg: Tuple[Any, Any]) -> Any:
    distance, leg = dist_leg
    return leg

long = unwrap(max(wrap(trip)))
short = unwrap(min(wrap(trip)))
```

不同于前一种实现方法，这种处理方式保留了路径段的完整信息：不仅提取了距离信息，而且把距离作为包装元组的第一项，然后利用 max()和 min()的默认行为模式处理包含距离和路径段的元组，最后舍弃第一个元素，保留路径段信息。

结果如下所示：

```
((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)
((35.505665, -76.653664), (35.508335, -76.654999), 0.1731)
```

最后且最重要的实现方法是使用 max()和 min()的高阶函数模式。首先定义一个辅助函数，然后用它递归路径段集合，提取需要的信息，代码如下所示：

```
def by_dist(leg: Tuple[Any, Any, Any]) -> Any:
    lat, lon, dist = leg
    return dist

long = max(trip, key=by_dist)
short = min(trip, key=by_dist)
```


函数 by_dist()拆开路径段元组，只返回其中的距离数据，距离数据后续将用于 max()函数和 min()函数。

max()和 min()都接收一个可迭代对象和和一个函数作为参数。在所有 Python 高阶函数中，都用关键字参数 key 来提取所需的关键字信息。

max()函数对 key 函数的使用如下所示：

```
from typing import Iterable, Any, Callable

def max_like(trip: Iterable[Any], key: Callable) -> Any:
    wrap = ((key(leg), leg) for leg in trip)
    return sorted(wrap)[-1][1]
```

可以理解为 max()函数和 min()函数用 key 函数把每一项包装成一个二元组。将二元组序列排序后，第一个值对应包含最小值的二元组，最后一个值对应包含最大值的二元组，拆包后就可得到原始数据。

其中的 key()函数是可选参数，默认值为 lambda x: x。

5.2 使用 Python 匿名函数

很多情况下，编写辅助函数需要太多代码。key 函数往往只是简单的表达式，这时候还需要写 def 和 return 语句就太烦琐了。

使用 Python 匿名函数有助于简化高阶函数的编写。顾名思义，匿名函数是没有名字的函数。采用 lambda 形式，函数体只能是简单的表达式。

使用 lambda 表达式作为键的示例如下：

```
long = max(trip, key=lambda leg: leg[2])
short = min(trip, key=lambda leg: leg[2])
```

lambda 从序列中获取输入。每个 leg 三元组作为该匿名函数的输入。对表达式 leg[2]进行求值，从三元组中提取距离数据。

理想状态下，只能使用一次匿名函数，但实际情况是经常需要复用匿名函数。复制粘贴当然不是好办法，有什么别的方法可以解决复用问题吗?

可以把匿名函数赋给变量，如下所示：

```
start = lambda x: x[0]
end = lambda x: x[1]
dist = lambda x: x[2]
```

一个匿名函数实际上是一个可调用对象，可以用作函数。

在交互式命令行中使用匿名函数的示例如下：

```
>>> leg = ((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)
>>> start = lambda x: x[0]
>>> end = lambda x: x[1]
>>> dist = lambda x: x[2]
>>> dist(leg)
129.7748
```

Python 提供了两种方法为元组元素命名：命名元组和匿名函数。二者功能相同，可以用匿名函数代替命名元组。

在此基础上，下面介绍如何从起点和终点提取经纬度数据。要实现这个目标，需要再定义一些匿名函数。

以下代码接着前面的交互式命令行：

```
>>> start(leg)
(27.154167, -80.195663)

>>> lat = lambda x: x[0]
>>> lon = lambda x: x[1]
>>> lat(start(leg))
27.154167
```

与命名元组相比，使用匿名函数抽取字段的优势似乎并不明显，而且用一组匿名函数提取字段要写的代码反而更多。但使用匿名函数让我们可以使用前缀式语法，在函数式编程语境中可读性更好。更重要的是，在后面 sorted()的例子中会看到，在 sorted()、max()和 min()函数中，使用匿名函数比使用命名元组效率更高。

5.3 lambda 与 lambda 算子

讨论纯函数编程的书一定会介绍 lambda 算子。这项技术由 Haskell Curry 发明，所以我们称它为“柯里化”。但 Python 并不特别依赖 lambda 算子，函数也不通过“柯里化”而归约为单参数匿名函数。

Python 匿名函数不受单参数限制，可以有多个参数，但函数体只能包含一个表达式。

第 10 章会详细介绍如何使用 `functools.partial` 实现“柯里化”。

5.4 使用 `map()`将函数应用于集合

标量函数将数值从定义域映射到值域。例如 `math.sqrt()`函数，它将浮点数 `x` 映射为另一浮点数 `y = sqrt(x)`，并且满足 $y^2=x$，定义域是所有正实数。这里的映射关系可以通过计算或者插值查表完成。

`map()`函数的作用与之类似：它将一个集合映射为另一个集合，保证将输入集合中的每个元素从定义域映射到值域中。这是使用内置函数处理集合的一种理想方式。

第一个例子是解析一段文本，获取一系列数值。假设现有如下文本段：

```
>>> text= """\
...   2    3    5    7   11   13   17   19   23   29
...  31   37   41   43   47   53   59   61   67   71
...  73   79   83   89   97  101  103  107  109  113
... 127  131  137  139  149  151  157  163  167  173
... 179  181  191  193  197  199  211  223  227  229
... """
```

用以下生成器函数重组这段文本：

```
>>> data = list(
...      v for line in text.splitlines()
...          for v in line.split())
```

首先把文本按行拆分，再对每一行使用空格进行二次拆分，结果如下所示：

```
['2', '3', '5', '7', '11', '13', '17', '19', '23', '29',
 '31', '37', '41', '43', '47', '53', '59', '61', '67', '71',
 '73', '79', '83', '89', '97', '101', '103', '107', '109', '113',
 '127', '131', '137', '139', '149', '151', '157', '163', '167',
 '173', '179', '181', '191', '193', '197', '199', '211', '223',
 '227', '229']
```

现在需要对序列中每个字符串元素使用 `int()`函数，可以借助 `map()`函数来实现，如下所示：

```
>>> list(map(int, data))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131,
 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197,
 199, 211, 223, 227, 229]
```

map()函数把 int()函数应用于序列的每个元素，返回结果是数值列表，而不是字符串列表。

map()函数的返回结果是可迭代对象，它能处理任何类型的可迭代对象。

可以通过 map()函数将任何 Python 函数应用于集合。很多内置函数可以应用于这样的“映射–处理”场景中。

在 map()中使用匿名函数

假设现在需要把旅行数据中的距离从海里转换为英里，也就是给每个路径段的距离字段乘以 6076.12 / 5280，即 1.150780。

可以使用 map()函数完成计算：

```
map(
    lambda x: (start(x), end(x), dist(x) * 6076.12 / 5280),
    trip
)
```

map()函数将作为参数的匿名函数应用于旅行数据的每个路径段，该匿名函数又会使用其他匿名函数分离路径段中的起点、终点和距离字段，最终计算出以英里为单位的距离值，然后重新组合成包含起点、终点和英里距离的元组。

此过程与下面的生成器表达式完全一致：

```
((start(x), end(x), dist(x) * 6076.12 / 5280) for x in trip)
```

这样就用生成器表达式完成了相同的处理。

二者的重要区别是：map()函数可以复用已有的函数定义或者匿名函数，更好的方法如下所示：

```
to_miles = lambda x: start(x), end(x), dist(x) * 6076.12 / 5280
trip_m = map(to_miles, trip)
```

这种实现方式将变换逻辑 to_miles 从对数据的处理过程中分离了出来。

5.5 使用 map()函数处理多个序列

有时需要把多个序列放在一起处理。第 4 章介绍过 zip()函数可以将两个序列组合在一起，生成新的对序列（二元组序列）。很多时候需要对序列做如下处理：

```
map(function, zip(one_iterable, another_iterable))
```

首先基于两个可迭代序列创建新参数元组，在把一个函数应用于该元组，如下所示：

```
(function(x, y)
    for x, y in zip(one_iterable, another_iterable)
)
```

这里用生成器表达式代替了 map()函数。

抽象前面的处理过程，如下所示：

```
def star_map(function, *iterables)
    return (function(*args) for args in zip(*iterables))
```


实际上，Python 提供了一个更好的解决方案，我们不必运用上面的技术了。下面举例说明如何使用这个方法。

第 4 章从 XML 文件中提取了一系列代表旅行路线的路径点信息，当时需要基于这一系列点数据创建包含起点和终点的路径段。

使用 zip()函数处理可迭代对象的一个简化版本如下：

```
>>> waypoints = range(4)
>>> zip(waypoints, waypoints[1:])
<zip object at 0x101a38c20>
>>> list(_)
[(0, 1), (1, 2), (2, 3)]
```

这样就基于一维列表创建出了对序列，每个数据对包含相邻的两个点。较短的序列元素用完之后 zip()函数的组合操作即结束。zip(x, x[1:])模式只能用于实例化的序列，以及由 range()函数创建的可迭代对象。

创建数据对以便用 haversine()函数计算路径上两点间的距离，其实现方法如下：

```
from ch02_ex3 import (lat_lon_kml, float_from_pair, haversine)

path = tuple(float_from_pair(lat_lon_kml()))
distances_1 = map(
    lambda s_e: (s_e[0], s_e[1], haversine(*s_e)),
    zip(path, path[1:])
)
```

首先把路径点信息加载到 path 变量中，实际上是一系列有序的经纬度数值对。由于下面要运用 zip(path, path[1:])设计模式，所以变量只能是实例化的序列，不能是可迭代对象。

zip()函数的返回结果是包含起点、终点的二元组序列，而我们需要的是包括起点、终点和距离的三元组序列。这里的匿名函数从输入二元组中提取数据，计算出距离后组合生成三元组。

前面提过，可以用 map()函数的高级形式将这一步简化为：

```
distances_2 = map(
    lambda s, e: (s, e, haversine(s, e)),
    path, path[1:])
```

请注意，map()函数的参数包括一个函数和两个可迭代对象。map()函数从两个可迭代对象中分别取出当前值，作为上面函数的输入参数。对于本例，这个指定的函数是一个返回包含起点、终点和距离的三元组的匿名函数。

map()函数的正式定义指出，它可以使用 star-map 方法处理任意多个可迭代对象。它从每个可迭代对象中取出当前值，作为指定函数的参数。

5.6　使用 filter()函数接收或舍弃数据

filter()函数的作用是把一个判定函数（也称“谓词函数”）应用于集合中的每个值。如果谓词为真，接收该值，否则将其舍弃。itertools 模块中的 filterfalse()函数是 filter()函数的一个变体，第 8 章会介绍 itertools 模块中 filterfalse()函数的用法。

下面利用 filter()函数过滤旅行数据，生成一个路径段长度大于 50 海里的子集：

```
long = list(
    filter(lambda leg: dist(leg) >= 50, trip))
)
```

对于长路径段，谓词函数返回 True，这段路径通过测试，短路径则不能通过测试。最终有 14 个路径段通过了长度测试。

这种处理方法很好地区分了过滤规则(lambda leg: dist(leg) >= 50)和其他处理过程，例如创建 trip 对象和分析长距离路径段等。

另一个例子如下所示：

```
>>> filter(lambda x: x % 3 == 0 or x % 5 == 0, range(10))
<filter object at 0x101d5de50>
>>> sum(_)
23
```

这里创建了一个简单的匿名函数，测试输入值是否为 3 或者 5 的倍数，然后把它应用于一个可迭代对象 range(10)，返回一个由通过测试的数值组成的可迭代序列。

谓词函数为真的数包括[0, 3, 5, 6, 9]，所以这些数据通过了测试，而谓词为假的数据都被舍弃了。

这个处理过程也可以用生成器表达式来完成，如下所示：

```
>>> list(x for x in range(10) if x % 3 == 0 or x % 5 == 0)
[0, 3, 5, 6, 9]
```

此处理过程的严格数学表述如下所示：

$$\{x \mid 0 \leqslant x < 10 \wedge (x \bmod 3 = 0 \vee x \bmod 5 = 0)\}$$

公式表明由 x 组成的集合满足在范围 `range(10)` 中，并且 `x % 3 == 0` 或者 `x % 5 == 0`，可以看到 `filter()` 函数和公式形式的数学表述之间存在很好的对应关系。

`filter()` 函数中的谓词函数经常需要使用已定义的函数，而不是匿名函数。复用之前定义过的函数作为谓词函数的示例如下：

```
>>> from ch01_ex1 import isprimeg
>>> list(filter(isprimeg, range(100)))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
71, 73, 79, 83, 89, 97]
```

这里从另一个模块中导入了一个名为 `isprimeg()` 的函数，然后将其应用于一个数值集合，保留质数，舍弃非质数。

用这种方法生成质数表的效率很低，它表面上的简洁类似于律师们常说的“保护儿童免受危险物品侵害原则”，看上去很好，实际上扩展性很差。`isprimeg()` 函数在计算每个值时都要重设测试条件，必须通过某种方法保存之前质数测试的计算结果。更好的方法是使用**埃拉托斯特尼筛法**（Sieve of Eratosthenes），该算法保留了之前得出的质数，因而可以避免大量低效的重复计算。

5.7 使用 filter() 函数检测异常值

前面定义了几个统计函数，用于计算平均值、标准差以及对数据进行归一化。利用这些函数查找数据中的异常值时，首先把 `mean()` 函数和 `stdev()` 函数应用于旅行数据中路径段里的距离值，计算总体平均值和标准差。

然后利用 `z()` 函数计算每个路径段的归一值。如果归一值大于 3，说明该样本远离平均值。去掉这些值可以提高数据的一致性，降低因测量误差引起的计算偏差。

具体实现方法如下：

```
from ch04_ex4 import mean, stdev, z

dist_data = list(map(dist, trip))
μ_d = mean(dist_data)
σ_d = stdev(dist_data)
outlier = lambda leg: z(dist(leg), μ_d, σ_d) > 3
print("Outliers", list(filter(outlier, trip)))
```

首先把距离计算函数映射到旅行数据的每个路径段上，由于后面要多次用到这个计算结果，

所以必须把它实例化为一个列表对象。这里不能用迭代器，因为第一个使用它的函数会把它包含的数据消耗掉。然后基于上面的计算结果求总体统计值 μ_d 和 σ_d，即平均值和标准差。

基于这些统计值，就可以用检查异常的匿名函数过滤数据，太大的归一值就是我们要找的异常值。

list(filter(outlier, trip))的计算结果是由两个路径段组成的列表，这两段比其他路径段大得多。路径段的平均长度是 34 海里，标准差是 24 海里，归一化距离没有小于–1.407 的。



应把复杂的问题拆解为一系列独立函数，每个函数都可以单独测试，以保证整个处理过程是由一系列简单函数构成的，这才是简洁明了的函数式编程。

5.8　在 iter()函数中使用哨兵值

Python 内置的 iter()函数把集合对象转换为迭代器，list,dict 和 set 类都可以通过 iter()函数转换成基于自己集合的迭代器对象。大多数情况下，使用 for 语句进行隐式转换，但在一些特殊情况下，需要显式创建迭代器，例如把集合的头部与后面分开。

其他场景包括用迭代器读取可调用对象（例如函数）的返回值，直到匹配到哨兵值。可以把它与 read()函数结合来读取文件，直到遇到行结束符或者文件结束符。例如 iter(file.read, '\n')会一直读取文件内容，直到遇到哨兵值'\n'。使用它的时候要注意，如果在数据中一直没有遇到哨兵值，函数将一直反复读取零长度的字符串。

作为 iter()参数的函数必须维护内部状态，使用起来有一定难度。函数式编程往往避免维护内部状态，然而所有的打开文件对象都会维护一个外界不可见的状态，比如 read()函数或者 readline()函数的内部状态是下一个字符或者下一行的位置。

显式迭代的另一种用法是可变集合使用 pop()方法改变自身状态，使用 pop()方法的示例如下：

```
>>> tail = iter([1, 2, 3, None, 4, 5, 6].pop, None)
>>> list(tail)
[6, 5, 4]
```

tail 变量中保存的是列表[1, 2, 3, None, 4, 5, 6]使用 pop()方法生成的迭代器。pop()方法默认弹出下标为–1 的列表元素，也就是按照逆序依次弹出各个元素。每次执行 pop()方法时，移除一个元素，列表内容发生变化，列表对象状态随之变化。当匹配到哨兵值后，迭代器不再返回数据，如果一直没有匹配到哨兵值，IndexError 异常会终止函数的执行。

应尽量避免维护对象的内部状态，本书不再涉及这一语言特征。

5.9　使用 sorted()函数将数据排序

当需要将按照指定规则将数据排序时，有两种方法可用。可以创建列表对象，然后用 list.sort()方法将这个列表排序。或者选用 sorted()函数，可以用它处理任何可迭代对象，返回排序后的列表对象。

sorted()函数有两种用法：直接应用于集合，或者作为高阶函数使用 key 参数定制排序方法。

下面处理第 4 章的旅行数据。之前编写的函数生成了一个由起点、终点和距离组成的三元组序列，代表旅行线路，数据如下所示：

```
(
 ((37.54901619777347, -76.33029518659048), (37.840832, -76.273834), 17.7246),
 ((37.840832, -76.273834), (38.331501, -76.459503), 30.7382),
 ((38.331501, -76.459503), (38.845501, -76.537331), 31.0756),
 ((36.843334, -76.298668), (37.549, -76.331169), 42.3962),
 ((37.549, -76.331169), (38.330166, -76.458504), 47.2866),
 ((38.330166, -76.458504), (38.976334, -76.473503), 38.8019)
)
```

使用如下交互式命令行查看 sorted()函数的默认行为：

```
>>> sorted(dist(x) for x in trip)
[0.1731, 0.1898, 1.4235, 4.3155, ... 86.2095, 115.1751, 129.7748]
```

这里首先使用生成器表达式(dist(x) for x in trip)从旅行数据从提取距离信息，然后将生成的可迭代对象排序，得到了从约 0.17 海里到约 129.77 海里的距离序列。

如果想在返回结果中保留起点、终点和距离，即原来的三元组形式，可以使用 sorted()函数并结合 key 函数指明排序的方式，如下所示：

```
>>> sorted(trip, key=dist)
[
 ((35.505665, -76.653664), (35.508335, -76.654999), 0.1731),
 ((35.028175, -76.682495), (35.031334, -76.682663), 0.1898),
 ((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)
]
```

这样就按照距离把旅行数据排序好了，其中排序函数 dist 如下所示：

```
dist = lambda leg: leg[2]
```

这个例子很好地展示了可以使用匿名函数从复杂元组中提取待排序数据。

5.10　编写高阶函数

高阶函数可以分为以下三类。

- 以函数为参数的函数。
- 以函数为返回值的函数，例如 Callable 类。也可以将返回生成器表达式的函数视为高阶函数。
- 以函数为参数，并且返回函数的函数，例如 functools.partial()，第 10 章将介绍。装饰器是另一种情况，第 11 章将介绍。

我们将使用高阶函数处理简单形式的数据，改变数据的结构，以及执行如下变换：

- 打包数据，生成更复杂的对象；
- 从复杂数据对象中抽取某个组件；
- 平铺复杂数据序列；
- 结构化一维数据序列。

Callable 类对象常用于那些返回可调用对象的函数，后面会利用它编写能注入配置参数的函数。

本章还会介绍简单的装饰器，第 11 章会深入研究装饰器相关问题。

5.11　编写高阶映射和过滤函数

Python 的两个内置高阶函数 map()和 filter()基本上可以处理任何数据。总的说来，优化这两个函数以获得更高性能比较困难。第 8 章会介绍 Python 3.4 的几个函数，包括 imap()、ifilter()和 ifilterfalse()等。

表达映射关系有 3 种方式，假设有函数 f(x)，集合对象 C，可以用下面 3 种方式表达映射关系：

- map()函数：map(f, C)
- 生成器表达式：(f(x) for x in C)
- 包含 yield 语句的生成器函数：

```
def mymap(f, C):
    for x in C:
        yield f(x)
mymap(f, C)
```

类似地，可以用下面 3 种方法把 filter()函数应用于集合。

- filter()函数：filter(f, C)。
- 生成器表达式：(x for x in C if f(x))。
- 包含 yield 语句的生成器函数：

```
def myfilter(f, C):
    for x in C:
        if f(x):
            yield x
myfilter(f, C)
```

不同方式的性能存在一定差异，其中 map() 和 filter() 最快。更重要的是，可以用不同的方式扩展上面的映射和过滤，如下所示。

- 可以创建一个更复杂的函数 g(x) 应用于每个元素，或者在处理前先对集合应用一个函数。这些方法对上面三种设计模式都适用，也是函数式设计擅长的领域。
- 修改生成器表达式或者生成器函数中的 for 循环。比较常用的处理方法是为生成器表达式添加 if 从句，从而将映射和过滤合并到一次操作中。还可以合并 mymap() 函数和 myfilter() 函数，从而实现合并映射操作和过滤操作。

在软件不断完善和成熟的过程中，常通过修改循环体来调整数据结构。对此有多种设计模式，包括打包、拆包（或者展开）、平铺以及结构化，之前介绍过其中几种。

设计映射时，如果一个函数中包含太多转换，就要注意了。应尽量避免创建那些主要功能不明的函数。由于 Python 没有优化编译器，有时必须通过合并函数来手动优化一些低性能的应用。但只在别无他法时才可以使用这类优化，也就是说，只有在性能分析表明应用需要优化时，才进行优化。

5.11.1 拆包并映射数据

当使用类似于 (f(x) for x, y in C) 这样的结构时，我们使用 for 语句中的多重赋值拆开一个元组，并将一个函数应用于其中一部分数据。整体上这是一个映射。这是 Python 常用的改变结构和应用函数的优化方法。

继续使用第 4 章中的旅行数据。下面是一个拆包并映射的具体实现。

```
from typing import Callable, Iterable, Tuple, Iterator, Any

Conv_F = Callable[[float], float]
Leg = Tuple[Any, Any, float]

def convert(
        conversion: Conv_F,
        trip: Iterable[Leg]) -> Iterator[float]:
    return (
        conversion(distance) for start, end, distance in trip
    )
```

该高阶函数使用下面的转换函数处理原始数据。

```
to_miles = lambda nm: nm * 5280 / 6076.12
to_km = lambda nm: nm * 1.852
to_nm = lambda nm: nm
```

有了转换函数，就可以提取距离数据，再进行转换了，如下所示：

```
convert(to_miles, trip)
```

处理结果是一系列浮点数，如下所示：

```
[20.397120559090908, 35.37291511060606, ..., 44.652462240151515]
```

从 `convert()`函数的 `for` 语句不难看出，这个函数只适用于“起点–终点–距离”格式的旅行数据。

可以进一步抽象这类“拆包–映射”设计模式，整个过程稍复杂。首先需要定义通用分解函数，如下所示：

```
fst = lambda x: x[0]
snd = lambda x: x[1]
sel2 = lambda x: x[2]
```

要表达 `f(sel2(s_e_d)) for s_e_d in trip`，就要用到复合函数。我们把一个类似于 `to_miles()`的函数和一个类似于 `sel2()`的选择器组合在一起。在 Python 中，可以用匿名函数把函数组合在一起，如下所示：

```
to_miles = lambda s_e_d: to_miles(sel2(s_e_d))
```

这样就得到了一个更长，但更通用的实现，如下所示：

```
(to_miles(s_e_d) for s_e_d in trip)
```

这个版本更具通用性，不过用处似乎不太大。

对于上述 `convert()`高阶函数，需要特别注意的是，它使用函数作为参数，并返回生成器函数作为结果。`convert()`函数不是生成器函数，不会生成任何值，它的返回值是一个生成器表达式，需要对其求值才能得到具体的计算结果。我们使用 `Iterator[float]`来强调返回结果是迭代器，也就是 Python 生成器函数的子类。

在这个设计模式中，可以用多重过滤代替映射，实现方法是在返回的生成器表达式中，用 `if` 从句实现过滤功能。

映射和过滤结合可以生成更复杂的函数。创建复杂的函数来缩短处理过程似乎是个好主意，但实际情况并非总是如此。复杂函数的性能往往不如嵌套的 `map()`函数和 `filter()`函数。总体而言，只有在函数通过封装概念来降低软件开发复杂度的情况下，才考虑使用复杂函数。

5.11.2 打包多项数据并映射

当使用类似于`((f(x), x) for x in C)`这样的结构时，首先将多个值打包以创建元组，然后进行映射。通常用这种方法保存中间计算结果。它既可以避免使用包含复杂状态的对象，也可以避免重复计算。

下面是第 4 章中根据路径点创建旅行数据的计算过程的一部分，代码如下所示：

```
from ch02_ex3 import (
    float_from_pair, lat_lon_kml, limits, haversine, legs
)

path = float_from_pair(float_lat_lon(row_iter_kml(source)))
trip = tuple(
    (start, end, round(haversine(start, end), 4))
    for start, end in legs(iter(path))
)
```

对上述代码稍做修改，创建一个高阶函数把打包部分与其他函数分开，具体实现如下：

```
from typing import Callable, Iterable, Tuple, Iterator

Point = Tuple[float, float]
Leg_Raw = Tuple[Point, Point]

Point_Func = Callable[[Point, Point], float]
Leg_D = Tuple[Point, Point, float]

def cons_distance(
        distance: Point_Func,
        legs_iter: Iterable[Leg_Raw]) -> Iterator[Leg_D]:
    return (
        (start, end, round(distance(start,end), 4))
        for start,end in legs_iter
    )
```

这个函数将每个路径段拆解开，保存在`start`和`end`两个变量里，这些变量是类型`Point`的实例，也就是由浮点数组成的二元组。这些变量作为`distance()`函数的参数，计算出每段距离。距离函数接收两个`Point`类型对象作参数，返回一个`float`值。`cons_distance()`返回一个三元组，包括原始数据中的起点和终点，以及距离计算结果。

接下来修改`trip`的赋值表达式，通过`haversine()`函数计算距离，如下所示：

```
path = float_from_pair(float_lat_lon(row_iter_kml(source)))
trip2 = tuple(cons_distance(haversine, legs(iter(path))))
```

用高阶函数`cons_distance()`代替了生成器表达式，使用函数作为参数，返回生成器表达式。

对上面的函数稍做修改，得到如下函数：

```
from typing import Callable, Iterable, Tuple, Iterator, Any
Point = Tuple[float, float]
Leg_Raw = Tuple[Point, Point]
Point_Func = Callable[[Point, Point], float]
Leg_P_D = Tuple[Leg_Raw, ...]

def cons_distance3(
        distance: Point_Func,
        legs_iter: Iterable[Leg_Raw]) -> Iterator[Leg_P_D]:
    return (
        leg + (round(distance(*leg), 4), ) # 1-tuple
        for leg in legs_iter
    )
```

该版本清晰地展示了如何基于原有对象创建新的对象，遍历旅行数据中的每个路径段，类型为 `Leg_Raw`，即包含两个点的二元组，沿着路径段计算距离，把代表原始路径段的 `Leg_Raw` 类型的对象与计算得到的距离组合在一起。

由于这些 `cons_distance()` 函数都接收函数作为参数，因此可以用其他公式计算距离。例如，可以使用 `math.hypot(lat(start) - lat(end), lon(start) - lon(end))` 计算出每个路径段起点和终点间的平面距离，尽管这种算法不太合适在这里使用。

第 10 章将使用 `partial()` 函数设置 `haversine()` 函数中 `R` 参数的值来改变距离计算单位。

5.11.3　平铺数据并映射

第 4 章介绍了将嵌套元组平铺为简单可迭代对象的算法。当时的目标只是改变数据结构，没有对数据做实质性处理，下面介绍如何将处理函数和平铺操作结合起来。

假设需要把如下文本转换为一维数值序列：

```
>>> text= """\
...   2   3   5   7  11  13  17  19  23  29
...  31  37  41  43  47  53  59  61  67  71
...  73  79  83  89  97 101 103 107 109 113
... 127 131 137 139 149 151 157 163 167 173
... 179 181 191 193 197 199 211 223 227 229
... """
```

每行包含 10 个数据，接下来要把所有行合并为一个数值序列。

通过下面这个由两部分组成的生成器函数实现数据转换：

```
data = list(
    v
    for line in text.splitlines()
    for v in line.split()
)
```

首先将整个文本分为多行，遍历每一行。在一行中，再将文本分为许多词组，输出是一个字符串列表，如下所示：

```
['2', '3', '5', '7', '11', '13', '17', '19', '23', '29', '31', '37',
 '41', '43', '47', '53', '59', '61', '67', '71', '73', '79', '83',
 '89', '97', '101', '103', '107', '109', '113', '127', '131', '137',
 '139', '149', '151', '157', '163', '167', '173', '179', '181', '191',
 '193', '197', '199', '211', '223', '227', '229']
```

除了改变格式，为了将字符串转换为数值，还需要增加一个转换函数，如下所示：

```
from numbers import Number
from typing import Callable, Iterator

Num_Conv = Callable[[str], Number]

def numbers_from_rows(
        conversion: Num_Conv, text: str) -> Iterator[Number]:
    return (
        conversion(value)
        for line in text.splitlines()
        for value in line.split()
    )
```

该函数有个 conversion 参数，作为转换函数，处理前面平铺算法生成的每个值。

如下所示使用 numbers_from_rows()函数：

```
print(list(numbers_from_rows(float, text)))
```

这里使用内置函数 float()将一段文本转换为了一个浮点数列表。

可以将高阶函数和生成器表达式相结合来改写上面的函数，如下所示：

```
map(float,
    value
    for line in text.splitlines()
    for value in line.split()
)
```

这种实现方法有助于我们理解算法的整体结构。采用**组块**原则，函数的具体实现通过一个有意义的名字进行抽象，然后将其用于新的上下文中。虽然通常使用高阶函数，但某些情况下使用生成器表达式能使算法更简便。

5.11.4　过滤并结构化数据

前面 3 个例子都是将数据处理和映射相结合，过滤并处理数据相比而言不那么明了，下面举例说明过滤并处理这对组合虽然很有用，但是其应用场景不如处理并映射这对组合典型。

第 4 章介绍过，借助结构化数据算法，可以方便地将带有结构化算法的过滤器与一个复杂函数结合起来。从可迭代对象中取出部分值的函数如下：

```
from typing import Iterator, Tuple

def group_by_iter(n: int, items: Iterator) -> Iterator[Tuple]:
    row = tuple(next(items) for i in range(n))
    while row:
        yield row
        row = tuple(next(items) for i in range(n))
```

该函数从一个可迭代对象中取出 n 个元素，输入可迭代对象中的元素作为输出结果的一部分依次被 `yield` 语句返回。原则上，该函数递归处理输入可迭代对象中的值，但由于递归在 Python 中效率较低，这里用显式的 `while` 循环代替递归。

如下所示使用该函数：

```
group_by_iter(7,
    filter(lambda x: x % 3 == 0 or x % 5 == 0, range(100))
)
```

首先使用 `range()` 函数创建一个可迭代对象，然后用 `filter` 函数进行过滤，最后进行分组。

将分组和过滤放在同一个函数体中，就实现了一个函数包含分组和过滤两个处理步骤，修改后的函数如下所示：

```
def group_filter_iter(
        n: int, pred: Callable, items: Iterator) -> Iterator:
    subset = filter(pred, items)
    row = tuple(next(subset) for i in range(n))
    while row:
        yield row
        row = tuple(next(subset) for i in range(n))
```

首先使用谓词函数处理输入可迭代对象，生成一个非严格的可迭代对象，所以不会立刻对 `data` 变量求值，而只在需要的时候才求值，这个函数的实现方法与前面的例子一样。

略微简化一下上下文，如下所示使用该函数：

```
group_filter_iter(
    7,
    lambda x: x % 3 == 0 or x % 5 == 0,
    range(1, 100)
)
```

只需一次函数调用，就可以过滤并提取数据。但将 `filter()` 函数和其他处理步骤放在一起往往不够清晰，大多数情况下，单独的 `filter()` 函数比组合函数更实用。

5.12　编写生成器函数

可以将许多函数简洁地写成生成器表达式的形式，比如之前讲过的映射和过滤。另外，也可以用 Python 内置的高阶函数（例如 `map()`或者 `filter()`）或者生成器函数实现这些函数。编写代码时，如果生成器函数中包含多条语句，要留意代码实现是否偏离了函数式编程的指导原则：对无状态的函数求值。

使用 Python 进行函数式编程，就像在刀锋上行走：一边是纯函数式编程，一边是命令式编程。我们需要仔细地辨别出那些无法通过纯粹的函数式编程实现的部分，使用命令式 Python 完成它，并把这些部分与其他函数式部分的代码隔离开。

5

代码逻辑必须使用 Python 语句实现时，就只能用生成器函数。下面罗列了一些不能使用生成器表达式的场景。

- 使用 `with` 语句处理外部资源时，第 6 章将详述文件解析问题。
- 循环条件比较灵活，不能用 `for` 语句，而需要用 `while` 语句时，例如 5.11.3 节中的例子。
- 循环中，由于满足特定条件，需要使用 `break` 语句或者 `return` 语句提前结束循环时。
- 使用 `try-except` 结构处理异常时。
- 包含内部函数定义时。第 1 章和第 2 章讲过这样的例子，第 6 章还会介绍这种情况。
- 复杂的 `if-elif` 分支语句。当需要处理的分支多于一种，无法用 `if-else` 表达时，分支语句会变得相对复杂。
- 以及那些不常用的 Python 语句，包括 `for-else`、`while-else`、`try-else` 和 `try-else-finally` 等，也都无法在生成器表达式中使用。

通常使用 `break` 语句提前结束集合处理过程。当遍历时遇到的元素满足指定的要求，就可以结束整个处理过程了，这类似于检测集合中是否存在拥有某种属性元素的 `any()`函数。还有一种情况是在处理完指定数量的元素（不是所有元素）后退出。

可以将寻找集合中的特定值简洁地表达为 `min(some-big-expression)`或者 `max(something big)`。在这种情况下，必须检查集合中的所有元素，确保选出的是最大值或者最小值。

少数情况下，需要实现一个 `first(function, collection)`函数，只要找到第一个值就够了。为了避免不必要的计算，遍历结束得越早越好。

下面是该函数的一种实现：

```
def first(predicate: Callable, collection: Iterable) -> Any:
    for x in collection:
        if predicate(x): return x
```

遍历集合，对集合中的每个元素应用指定的谓词函数。如果谓词函数判断结果为 `True`，则

返回对应的元素；如果集合遍历完毕，则返回默认的 None 值。

可以从 PyPI 上下载该函数的一个版本，first 模块基本上是对上面的思想的实现与扩展，更多细节可参考 https://pypi.python.org/pypi/first。

判断一个数是否为质数时就可以用这里的 first 函数，示例如下：

```
import math
def isprimeh(x: int) -> bool:
    if x == 2: return True
    if x % 2 == 0: return False
    factor = first(
        lambda n: x % n == 0,
        range(3, int(math.sqrt(x) + .5) + 1, 2))
    return factor is None
```

该函数处理了一些特殊情况，包括 2 是质数，而其他偶数都不是质数，然后用上面定义的 first()函数寻找指定集合中的第一个因子。

first()函数返回第一个因子。在这个场景中，这个数具体是什么并不重要，存在与否才重要。当因子不存在时，isprimeh()函数返回 True。

可以使用同样的方法处理数据异常。处理无效数据的 map()函数如下：

```
def map_not_none(func: Callable, source: Iterable) -> Iterator:
    for x in source:
        try:
            yield func(x)
        except Exception as e:
            pass # For help debugging, use print(e)
```

该函数遍历可迭代对象中的每个元素，并将函数应用于元素。如果没有异常，则返回处理结果；如果发现了异常，则舍弃该异常元素。

处理包含无效值或者缺失值的数据时，这种方法很方便，无须创建复杂的过滤器筛选异常值，只要在处理过程中舍弃无效的输入值即可。

可以对包含无效值的输入数据执行映射，如下所示：

```
data = map_not_none(int, some_source)
```

some_source 集合由字符串组成，map_not_none()函数对其中元素依次应用 int()函数，可以方便地过滤掉那些不代表数字的字符串。

5.13　使用可调用对象构建高阶函数

可以通过创建 Callable 类对象来定义高阶函数。与编写生成器函数的思路类似，编写可调

用对象是为了使用 Python 语句。除了能使用语句外，还可以在创建高阶函数时进行静态配置。

通过 class 声明定义 Callable 类对象，实际上定义了一个以函数为返回值的函数。通常可以使用可调用对象把两个函数组合起来，形成一个复杂函数。

如下面的类所示：

```
from typing import Callable, Optional, Any

class NullAware:
    def __init__(
            self, some_func: Callable[[Any], Any]) -> None:
        self.some_func = some_func
    def __call__(self, arg: Optional[Any]) -> Optional[Any]:
        return None if arg is None else self.some_func(arg)
```


这个类用于创建空值敏感的新函数。创建这个类的实例时需要提供一个函数（some_func）作为参数，对该函数的限制条件由 Callable[[Any], Any]定义，即输入单一值并返回单一值。返回结果是可调用的，并接收一个可选参数。__call__()方法处理参数为 None 的情况，这个方法将返回结果类型定义为 Callable[[Optional[Any]], Optional[Any]]。

对表达式 NullAware(math.log)求值，创建出一个作用于参数的新函数。__init__方法将用户定义的函数保存在结果对象中，该对象是个包含数据处理逻辑的函数。

通常会为生成的新函数指定一个名称，以便后续使用，如下所示：

```
null_log_scale = NullAware(math.log)
```

这里将新创建的函数赋给变量 null_log_scale，接下来就可以在新的上下文中使用该函数了，如下所示：

```
>>> some_data = [10, 100, None, 50, 60]
>>> scaled = map(null_log_scale, some_data)
>>> list(scaled)
[2.302585092994046, 4.605170185988092, None, 3.912023005428146,
4.0943445622221]
```

或者像下面这样，把创建和使用函数写在同一个表达式中：

```
>>> scaled = map(NullAware(math.log), some_data)
>>> list(scaled)
[2.302585092994046, 4.605170185988092, None, 3.912023005428146,
4.0943445622221]
```

对 NullAware(math.log)求值的返回结果是一个匿名函数，该函数在 map()函数中用于处理可迭代对象 some_data。

上面例子中的__call__()方法完全基于表达式求值，是一种简洁易用的基于底层组件函数

创建复合函数的方法。当使用标量函数时，只需考虑很少几个设计约束就可以了，而当涉及可迭代集合时，就需要仔细斟酌了。

确保正确的函数式设计

使用无状态的函数式 Python 代码时，要注意使用对象，因为对象是有状态的。实际上，面向对象编程旨在把状态变化封装到类定义中。这样在使用 Python 类定义处理集合时，你会发现函数式编程和命令式编程背道而驰。

使用可调用对象创建复合函数，让我们能以相对简单的语法使用这些复合函数。在使用可迭代映射或者归约时，需要注意引入有状态对象的原因及方式。

回到之前的 sum_filter_f() 复合函数。基于 Callable 类定义的实现如下：

```
from typing import Callable, Iterable

class Sum_Filter:
    __slots__ = ["filter", "function"]
    def __init__(self,
            filter: Callable[[Any], bool],
            func: Callable[[Any], float]) -> None:
        self.filter = filter
        self.function = func
    def __call__(self, iterable: Iterable) -> float:
        return sum(
            self.function(x)
            for x in iterable
                if self.filter(x)
        )
```

这个类的每个对象只能有两个属性，以降低后续把该函数当成有状态对象使用的可能性。这种限制并不能完全杜绝对返回结果对象的修改，但将对象属性限制为两个，使得添加其他属性将引发异常。

初始化方法 __init__() 为对象实例加载了两个函数：filter 和 func。__call__() 方法的返回值是一个生成器表达式，这个生成器表达式又使用了前两个函数。其中 self. filter() 方法用于对集合元素进行取舍，self.function() 函数用于转换通过了 filter() 函数检验的元素。

类的实例是包含两个策略函数的函数。创建类的实例如下所示：

```
count_not_none = Sum_Filter(
    lambda x: x is not None,
    lambda x: 1)
```

该函数的功能是计算序列中非 None 元素的个数。它使用了两个匿名函数：一个过滤出序列中的非 None 元素，另一个将所有非 None 元素转换为 1。

总的说来，这个 count_not_none()函数与其他 Python 函数一样，使用方法比前面的 sum_filter_f()函数简单一些。

如下所示使用 count_not_none()函数：

```
N = count_not_none(data)
```

如下所示使用 sum_filter_f()函数：

```
N = sum_filter_f(valid, count_, data)
```

可以看出，基于一个可调用对象的 count_not_none()函数比普通函数的参数要少，所以使用起来相对简单。但它定义函数行为的代码分散在两处（可调用类中的定义，以及使用函数时指定的参数），从这个角度看，这种方式似乎不够清晰明了。

5.14　设计模式回顾

在不提供 key 函数的情况下，函数 max()、min()以及 sorted()按照默认方式执行。当提供了 key 函数，则按照 key 函数指定的方法基于输入数据计算 key 值。在之前的例子中，计算 key 值的函数都是从输入中简单地提取数据，但其实不必如此，使用 key 函数可以做任何事。

比如函数 max(trip, key=random.randint())。不过总的说来，尽量避免用 key 函数做这类难以理解的事。

使用 key=函数是常见的设计模式，在编写函数的过程中可以方便地使用它。

前面介绍过可以使用匿名函数简化高阶函数的编写。使用匿名函数最大的好处是它遵循函数式编程范式。如果使用传统方法定义函数，很容易混入命令式编程代码，从而影响函数式设计的简洁性和可读性。

至此，已经介绍了处理集合数据的多种高阶函数。前一章详细分析了与集合对象或标量函数相关的几种设计模式。高阶函数的完整分类如下。

- **返回生成器**：返回生成器表达式的函数属于高阶函数，因为其返回值不是标量或者集合。某些高阶函数也能以函数为参数。
- **类生成器**：函数体中包含 yield 语句的函数，即所谓的“头等生成器函数”。生成器函数可以返回惰性求值的可迭代集合。实际上，生成器函数和返回生成器表达式的函数在功能上基本没有区别，都是非严格的，都返回一个值序列。因此，通常将生成器函数也视作高阶函数，内置函数 map()和 filter()就属于这一类。
- **实例化集合**：有些函数返回一个实例化的集合对象，例如列表、元组、set 以及映射等。如果该函数有至少一个函数参数，它就是高阶函数，否则它是处理集合的普通函数。

- **归约集合**：有些函数的输入是集合对象，输出是标量值，例如 len()和 sum()函数。当参数中包含函数时，也属于高阶函数，后面会详细讨论这个问题。
- **标量**：处理单个值的函数。如果输入参数中包含另一个函数，也属于高阶函数。

编写软件时，上述设计模式可供参考。

5.15　小结

本章首先介绍了两个高阶归约函数——max()和 min()，接着介绍了两个重量级高阶函数——map()和 filter()，还提到了 sorted()。

然后讨论了如何使用高阶函数改变数据的结构，介绍了针对不同类型数据的几种常用的变换方式：打包、拆包、平铺以及结构化。

最后介绍了定义高阶函数的 3 种方式：

- 使用 def 语句，以及与之类似的将匿名函数赋给变量的方式；
- 定义一个可调用类，实质上定义了一个能返回复合函数的函数；
- 使用装饰器创建复合函数，第 11 章将深入讨论该话题。

下一章将介绍如何通过递归实现纯粹的函数迭代，以及运用纯粹的函数式技术，并基于 Python 式的数据结构进行改进。下一章还会涉及通过归约把集合转换为标量值的相关问题。

第 6 章

递归与归约

6

前面介绍了多种互相关联的数据处理模式，如下所示：

- 对集合进行映射并过滤得到集合；
- 对集合进行归约得到标量值。

`map()`和 `filter()`属于第一类集合处理函数，很好地体现了两种模式的的区别。第二类归约函数包括`min()`、`max()`、`len()`、`sum()`，以及通用归约函数`functools.reduce()`。

也可以将 `collections.Counter()`函数看作归约操作。它的计算结果不是标量值，但它去掉了输入数据的某些信息，返回了新的结构体。本质上，它执行的操作是分组并计数，相较于映射，更近于归约。

本章将详细介绍归约函数。从纯函数的角度看，归约是通过递归定义的，因此在讲解归约算法之前，先介绍递归。

总体而言，函数式语言的编译器会通过将尾递归调用转换为循环来优化递归函数，进而大幅提升函数性能。对于 Python 语言，纯粹的递归是受限的，必须手动优化尾调用，也就是显式地把递归调用转换为 `for` 循环。

本章会介绍归约算法`sum()`、`count()`、`max()`和`min()`，还会介绍`collections.Counter()`函数以及相关的`groupby()`函数。解析（以及文本扫描）其实是一种归约，因为它们将标记序列（或者字符序列）转换成了包含复杂属性的高阶集合。

6.1　简单数值递归

所有的数值计算都能以递归的方式定义。可访问维基百科，了解皮亚诺公理的更多相关信息。

从这些公理中不难看出加法是通过一个更基本的概念，数字 n 的下一个数（也称“后继数”）$S(n)$，以递归的方式定义的。

为了便于理解，首先定义前置函数 $P(n)$，当 $n \neq 0$ 时，该函数满足 $n = S(P(n)) = P(S(n))$ 。

这样就可以通过递归的形式定义两个自然数之间的加法了，如下所示：

$$\text{add}(a,b) = \begin{cases} b & a = 0 \\ \text{add}(P(a), S(b)) & a \neq 0 \end{cases}$$

如果用更常见的 n+1 和 n–1 代替 $S(n)$和 $P(n)$，那么有 add(a,b)=add(a–1,b+1)。

可以将上述公式转换为 Python 代码，如下所示：

```
def add(a: int,b: int) -> int:
    if a == 0:
        return b
    else:
        return add(a - 1, b + 1)
```

只是按 Python 语法把数学符号重排了一下，无他。

实际上，无须用 Python 写函数实现加法计算，利用 Python 已有的功能就足以实现各种算术运算了。举这个例子是想说明能以递归的方式定义基本的标量计算，其实现也不复杂。

递归定义至少包括两种情况：非递归情形（也称“基础情形”），直接返回确定的值；递归情形，函数通过不同的输入参数对自身的调用给出返回值。

为了确保递归能结束，必须搞清楚输入值向非递归参数靠近的机制。前面定义的函数中虽然没写，但输入参数往往是有条件约束的。例如在前面定义的 `add()`函数中，输入值的约束是：`assert a >= 0 and b >= 0`。

如果没有这些约束，就不能保证 `a` 取–1 时函数向 `a == 0` 靠近。

6.1.1　实现尾调用优化

可以通过递归的形式把许多函数简洁明了地表达出来，其中最典型的例子是 `factorial()`。

用 Python 的递归形式表示如下：

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

对应的 Python 实现如下：

```
def fact(n: int) -> int:
    if n == 0: return 1
    else: return n * fact(n - 1)
```

该实现的主要优点是简洁,但受制于 Python 的递归嵌套数量限制,无法进行超过 fact(997)的计算。1000!有 2568 位数，超过了浮点数的最大值。在某些系统中这个上限是 10^{300}。编程实践中，经常使用 log gamma 函数处理过大的浮点数。

该函数是典型的尾递归形式，函数体的最后一个表达式是对使用了新参数的自身的调用。优化编译器能用循环代替函数调用栈，从而大幅提高计算速度。

由于 Python 没有优化编译器，所以必须考虑对它们进行优化。这里，函数从 n 向 n-1 变化，因此可以构造一个序列，通过归约计算乘积。

如果不考虑纯粹的函数式处理，可以定义如下所示的命令式计算流程：

```
def facti(n: int) -> int:
    if n == 0: return 1
    f = 1
    for i in range(2, n):
        f = f * i
    return f
```

此版本的实现能进行高于 1000!的计算（例如 2000!有 5733 位数）。它不是纯粹的函数式，我们用有状态的循环优化了尾递归，其中的 i 变量用于保存计算过程的状态。

总的说来，由于不能自动进行尾调用优化，在用 Python 编程时只能用这种方法。但有些情况下，这样的优化其实没有益处，下面会详述。

6.1.2 保持递归形式

有些情况下，递归形式的定义是最优解。一些递归实现通过分治策略极大地降低了工作量，例如通过平方法计算幂次函数，计算公式如下：

$$a^n = \begin{cases} 1 & n = 0 \\ a \times a^{n-1} & n \text{ 为奇数} \\ (a^{\frac{n}{2}})^2 & n \text{ 为偶数} \end{cases}$$

将整个计算过程划分为 3 种情况，用 Python 的递归形式实现如下：

```
def fastexp(a: float, n: int) -> float:
    if n == 0:
        return 1
    elif n % 2 == 1:
        return a * fastexp(a, n-1)
    else:
        t = fastexp(a, n // 2)
        return t * t
```

函数有 3 个分支。将基础型式 `fastexp(a, 0)`的返回值定义为 1，另外两个分支采用不同的处理策略，如果幂次为奇数，以递归形式定义返回值。幂次 `n` 减小 1，属于简单的尾递归优化情形。

如果幂次为偶数，则使用 `n/2` 计算返回值，问题规模缩小为原来的一半。由于问题规模是折半缩小的，所以处理速度会明显提升。

这种函数无法简单地改为尾调用优化循环。由于递归形式已经是最优解，无须进一步优化了。Python 的递归限制是 $n \leqslant 2^{1000}$，已经相当大了。

函数的类型标示表明参数类型为浮点数，并接收整型参数。根据 Python 的类型转换规则，当需要处理这两类数值时，使用浮点数作为函数类型标示更简单。

6.1.3　处理复杂的尾调用优化

可以用递归的形式定义斐波那契数列。第 n 个斐波那契数 F_n 的计算公式如下：

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geqslant 2 \end{cases}$$

将斐波那契数 F_n 定义为其前面两个数的和：$F_{n-1} + F_{n-2}$。这是多次递归的典型例子，无法用简单的尾递归优化进行转换。但如果不把它优化成尾递归，计算速度会慢到基本不可用。

下面是个简单的实现：

```
def fib(n: int) -> int:
    if n == 0: return 0
    if n == 1: return 1
    return fib(n-1) + fib(n-2)
```

由于是多次递归，当需要计算 `fib(n)`时，必须先算出 `fib(n - 1)`和 `fib(n - 2)`。由于计算 `fib(n - 1)`时重复计算了一次 `fib(n - 2)`，导致斐波那契函数的实际计算增加了一倍。

Python 是从左向右依次对表达式求值的，最大可以计算 `fib(1000)`，但实际计算这么大的数需要非常有耐心。

下面改写整个算法，用有状态的变量代替递归。

```
def fibi(n: int) -> int:
    if n == 0: return 0
    if n == 1: return 1
    f_n2, f_n1 = 1, 1
```

```
    for _ in range(3, n+1):
        f_n2, f_n1 = f_n1, f_n2+f_n1
    return f_n1
```



这里的有状态版本从 0 向上计算，而不像递归版本从 n 向下计算，速度会比递归版本快得多。

需要强调的是，对 fib()函数的递归优化没有简便方法。为了能用命令式版本代替递归版本，必须结合具体算法，确定所需中间变量的数量。

6.1.4 使用递归处理集合

也可以用递归的形式定义对集合的处理，例如用递归的形式定义 map()函数，公式如下：

$$\text{map}(f, C) = \begin{cases} [\ ] & \text{len}(c) = 0 \\ \text{map}(f, C[:-1]) \text{ append } f(C[-1]) & \text{len}(c) \neq 0 \end{cases}$$

对空集合的映射结果仍然是一个空序列，对非空集合，则是由 3 部分组成的递归形式的表达式。首先将函数应用于除最后一个元素外的集合，形成一个新序列，然后对集合最后一个元素应用函数，最后将返回结果追加到之前生成的序列上。

先前的 map()函数的纯递归实现如下：

```
from typing import Callable, Sequence, Any
def mapr(
        f: Callable[[Any], Any],
        collection: Sequence[Any]
    ) -> List[Any]:
    if len(collection) == 0: return []
    return mapr(f, collection[:-1]) + [f(collection[-1])]
```

map(f, [])的返回值是一个空列表对象，当输入值是非空列表时，首先将函数应用于列表的最后一个元素，然后将它追加到以递归形式定义的列表上，这个列表是函数应用于输入序列头部形成的。

需要说明的是，这里的 mapr()函数返回一个列表对象，类似于 Python 2 中老版本的 map()函数。Python 3 的 map()函数返回的是可迭代对象，而不是列表对象。

虽然这个实现很简洁，但没有尾调用优化。如果实现了尾调用优化，就能突破调用栈深度 1000 的限制，并大幅提升处理速度。

Callable[[Any], Any]是比较宽松的类型标示，用于定义领域类型变量和范围类型变量，后续的实例中会用到它。

6.1.5　集合的尾调用优化

处理集合大致有两种方法：使用高阶函数，返回一个生成器表达式，或者创建一个函数，用 for 循环依次处理集合中的每个值。这两种方法实际上很类似。

下面是一个类似于 Python 内置 map()函数的高阶函数：

```
from typing import Callable, Iterable, Iterator, Any, TypeVar
D_ = TypeVar("D_")
R_ = TypeVar("R_")
def mapf(
        f: Callable[[D_], R_],
        C: Iterable[D_]
    ) -> Iterator[R_]:
    return (f(x) for x in C)
```

返回结果是一个实现了映射处理的生成器表达式，用显式 for 循环实现了类似于尾调用优化的效果。

数据源 C 通过类型标示 Iterable[D_]强调映射域的类型。映射函数的类型标示是 Callable[[D_], R_]，表示将某个领域类型映射为某个范围类型。以 float()函数为例，它将字符串域中的值映射到浮点数范围内。返回结果的类型标示是 Iterator[R_]，是基于某个范围类型（即映射函数的结果类型）的迭代器。

实现了相同功能，具备相同类型签名的生成器函数如下：

```
def mapg(
        f: Callable[[D_], R_],
        C: Iterable[D_]
    ) -> Iterator[R_]:
    for x in C:
        yield f(x)
```

这里使用了完整的 for 语句来实现尾调用优化。返回结果与前面的实现一致。由于包含多条语句，运行速度略慢。

不论使用哪种方式，返回结果都是可迭代对象，必须通过某种方法从可迭代的数据源中实例化出一个序列对象，例如下面利用 list()函数实现实例化：

```
>>> list(mapg(lambda x: 2 ** x, [0, 1, 2, 3, 4]))
[1, 2, 4, 8, 16]
```

为了保证性能和伸缩性，需要对 Python 程序进行这类尾调用优化处理。虽然这种实现不是纯粹的函数式，但它带来的好处远抵过缺少纯粹性的损失。为了获得函数式设计简洁明了的优势，需要将这类不够纯粹的函数当作普通的递归使用。

这意味着在编程实践中应避免在集合处理函数中混入有状态的处理步骤（这会弄乱整个处理

流程）。这样即使处理流程的某些部分不是纯粹的函数式，从整个处理流程看，函数式编程的核心原则仍然适用。

6.1.6　集合的归约与折叠：从多个到一个

可以通过下面的方式定义 `sum()` 函数。

空集合的和为 0，非空集合的和是第一个元素加上剩余元素组成集合的和。

$$\text{sum}(C) = \begin{cases} 0 & \text{len}(C) = 0 \\ C[0] + \text{sum}(C[1:]) & \text{len}(C) > 0 \end{cases}$$

类似地，以递归方式计算集合乘积分以下两种情况：

$$\text{prod}(C) = \begin{cases} 1 & \text{len}(C) = 0 \\ C[0] \times \text{prod}(C[1:]) & \text{len}(C) > 0 \end{cases}$$

基础型式定义空集合的乘积为 1。递归型式定义非空集合的乘积是第一个元素乘剩余元素组成集合的乘积。

这样就定义了序列中每个元素间加法和乘法的 fold 操作，并且由于是从右向左处理集合元素，可以将上面的处理过程视作以 fold-righ 的方式将集合归约为标量值。

可以用递归的形式使用 Python 计算集合的乘积，如下所示：

```
def prodrc(collection: Sequence[float]) -> float:
    if len(collection) == 0: return 1
    return collection[0] * prodrc(collection[1:])
```

直接把数学定义转换为了 Python 代码，从技术上看完全正确，但这个实现方法不够理想，因为计算过程中会创建大量中间列表对象，而且只能处理集合，不能处理可迭代对象。

除此之外，类型标示中使用了 `float`，但也可以处理整型值并返回整型值。在这样的数值计算函数中，将 `float` 用作通用的类型标示更简便。

将其简单修改一下，就可以处理可迭代对象了，同时避免了创建任何中间对象。能处理迭代类型数据源的递归乘积函数如下所示：

```
def prodri(items: Iterator[float]) -> float:
    try:
        head = next(items)
    except StopIteration:
        return 1
    return head * prodri(items)
```

对可迭代对象，无法使用 `len()` 函数计算其元素数量，只能尝试从中取出第一个元素。如果

其中没有元素，尝试取元素将引发 `StopIteration` 异常。如果可迭代对象中有元素，将它与序列中剩余元素的乘积相乘。为了便于理解，这里显式地用 `iter()` 函数将一个实例化的序列转换为可迭代对象。实际编码过程中如果可迭代对象直接可用，就不必再转换了，示例如下：

```
>>> prodri(iter([1, 2, 3, 4, 5, 6, 7]))
5040
```

该递归实现不依赖显式状态，以及 Python 的其他命令式特性。虽然是纯粹的函数式，但它仍然受限于集合大小不能超过 1000。在具体的编程实践中，常用如下命令式结构实现递归函数：

```
def prodi(items: Iterable[float]) -> float:
    p = 1
    for n in items:
        p *= n
    return p
```

此版本不再受限于递归次数，实现了尾调用优化，并且能同时处理序列和可迭代对象。

在其他函数式语言中，把这个实现称为 `foldl`（即 `fold-left`）操作，因为它从左向右依次处理可迭代集合中的元素，与之相对的是 `foldr`（即 `fold-right`）操作，也就是从右向左依次处理集合中的元素。

对于实现了优化编译器和惰性求值的语言来说，`fold-left` 和 `fold-right` 的区别在于创建中间结果的方式。这一差异会影响计算性能，但通常不太明显。`fold-left` 首先处理序列的第一个元素，而 `fold-right` 会先取出第一个元素，但并不进行处理，而是待整个序列处理完后再处理此元素。

6.2　group-by 归约：从多到少

数据处理过程中常做的一种归约需要先按照某个键值或者指标对数据进行分组。在 SQL 中，通常称之为 `SELECT GROUP BY` 操作。原始数据按照某一列的值进行分组，然后对其他列的值进行归约（有时是累积）。SQL 累积函数包括 `SUM`、`COUNT`、`MAX` 和 `MIN`。

统计计算中的众数是对某个独立变量进行分组并计算其出现次数。首先用 Python 提供的方法对数据进行分组，然后对分组数据进行归约。下面先研究分组数据并计算频率的两种方法，再介绍如何求分组数据的其他特征值。

仍然使用第 4 章的旅行数据，这些数据由一系列包含经纬度的路径点组成。已经将其转换为了一系列包含起点、终点和距离的路径段，如下所示：

```
(((37.5490162, -76.330295), (37.840832, -76.273834), 17.7246),
 ((37.840832, -76.273834), (38.331501, -76.459503), 30.7382),
 ((38.331501, -76.459503), (38.845501, -76.537331), 31.0756), ...
 ((38.330166, -76.458504), (38.976334, -76.473503), 38.8019))
```

可以通过有状态映射或者实例化的有序对象来计算一组数据值的众数。旅行数据中的值是连续的，为了计算众数，需要将距离数据量子化，也称**分箱**：将数据分组，放入不同的箱中。分箱技术在数据可视化应用中也很常用，这里以 5 海里为单位设置箱子的大小。

可以用下面的生成器表达式计算量子化的距离：

```
quantized = (5 * (dist // 5) for start, stop, dist in trip)
```

每个距离值除以 5，丢掉余数，商再乘以 5，就得到了以 5 海里取整后的距离值。

6.2.1 用 Counter 做映射

当需要对集合按照某些值进行分组并计算频次时，可以使用 `collections.Counter` 等优化工具。对于函数式编程，分组数据的典型处理方式分为两步，首先对原始集合排序，然后用一个递归循环识别出每一组的开始位置。这当中涉及实例化原始数据，执行一个复杂度为 $O(n \log n)$ 的排序，最后对每个键值进行归约，得到累积值或者频次。

下面的生成器创建了分箱后的一组路径值：

```
quantized = (5 * (dist // 5) for start, stop, dist in trip)
```

每个距离值除以 5 取整再乘以 5，就得到了 5 海里的倍数。

下面的表达式创建了从距离到频次的映射：

```
from collections import Counter
Counter(quantized)
```

这是一个有状态的对象，从技术上看，它是通过命令式的面向对象编程创建出来的，但它看上去像函数，也符合函数式编程的设计理念。

打印 `Counter(quantized).most_common()` 函数的返回值，结果如下：

```
[(30.0, 15), (15.0, 9), (35.0, 5), (5.0, 5), (10.0, 5), (20.0, 5),
 (25.0, 5), (0.0, 4), (40.0, 3), (45.0, 3), (50.0, 3), (60.0, 3),
 (70.0, 2), (65.0, 1), (80.0, 1), (115.0, 1), (85.0, 1), (55.0, 1),
 (125.0, 1)]
```

出现频次最高的距离值是 30 海里，最短的路径段为 4 个 0 海里，最长的是 125 海里。

你的运行结果可能与上面的略有不同，因为 `most_common()` 函数是按频次排序的，相同的频次可能以任意顺序输出，所以下面 5 个长度的出现顺序可能会不同。

```
(35.0, 5), (5.0, 5), (10.0, 5), (20.0, 5), (25.0, 5)
```

6.2.2　用排序构建映射

如果不考虑 Counter 类，可以用更函数式的方式进行排序和分组。常用实现如下：

```
from typing import Dict, Any, Iterable, Tuple, List, TypeVar
Leg = Tuple[Any, Any, float]
T_ = TypeVar("T_")

def group_sort1(trip: Iterable[Leg]) -> Dict[int, int]:
    def group(
            data: Iterable[T_]
        ) -> Iterable[Tuple[T_, int]]:
        previous, count = None, 0
        for d in sorted(data):
            if d == previous:
                count += 1
            elif previous is not None: # and d != previous
                yield previous, count
                previous, count = d, 1
            elif previous is None:
                previous, count = d, 1
            else:
                raise Exception("Bad bad design problem.")
        yield previous, count
    quantized = (int(5 * (dist // 5)) for beg, end, dist in trip)
    return dict(group(quantized))
```

内部的 group() 函数遍历排序后的数据集合，如果当前值与前次循环值一致（等于 previous 中保存的值），给计数器加 1，如果不一致，并且前次循环值不是 None，说明新的键值出现了。这时首先返回前次循环值及其计数，然后创建新的键值对，键是当前值，值是计数器初始值。第三种情况只会出现一次：未设置前次循环值，这是循环的开始状态，应保存初始值。

group() 函数包含两个重要的类型标示，数据源是基于某种数据类型（用 T_ 表示）的可迭代对象，在这个例子中是整型，但适用于任何 Python 类型。由于返回结果中同样使用了类型变量 T_，可知 group() 函数的返回结果中包含的数据类型与数据源的类型一致。

group_sort1() 函数的最后一行基于分组后的数据创建了一个字典。与 Counter 创建的字典类似，主要区别在于 Counter 对象有 most_common() 方法，而普通的字典对象没有。

条件分支 elif previous is None 是比较笨拙的设计，去掉它难度不是太大（并且性能会略有提升）。

为了去掉该 elif 分支，需要调整一下 group() 函数的初始化方法。

```
def group(data: Iterable[T_]) -> Iterable[Tuple[T_, int]]:
    sorted_data = iter(sorted(data))
    previous, count = next(sorted_data), 1
    for d in sorted_data:
```

```
        if d == previous:
            count += 1
        elif previous is not None: # and d != previous
            yield previous, count
            previous, count = d, 1
        else:
            raise Exception("Bad bad design problem.")
    yield previous, count
```

首先从排序后的数据中取出第一个数据，作为 previous 的初始值，在循环中处理剩余的值。从这一设计思路中可以看到递归方法的使用：用序列的第一个值做递归的初始值，每次递归处理一个值，如果不是 None 则继续递归处理，否则说明数据已经处理完毕。

除此之外，用 itertools.groupby()也可以达到相同的效果，第 8 章将详细介绍该函数。

6.2.3 使用键值分组或者分区数据

对分组后的数据做何种归约是没有限制的。数据中可能包含一些自变量或者因变量。可以通过自变量对数据进行分区，然后计算每个分区的各项汇总值，包括最大值、最小值、平均值以及标准差等。

对数据做复杂归约的关键是保存每组中的所有数据。Counter 函数仅收集相同数据出现的频次，我们需要基于键值将原始数据转换为序列。

简而言之，每个 5 海里的箱子中都保存了该范围内路径段的所有数据，而不仅仅是出现次数。不妨将分区看作递归，或者 defaultdict(list)对象的有状态应用，下面介绍 groupby()函数的递归定义，它相对简单一些。

显然，groupby(C, key)函数对空集合 C 的返回值是空字典 dict()，或者一般而言，空 defaultdict(list)对象。

对于非空集合，首先处理集合头部 C[0]，然后递归处理集合尾部 C[1:]。可以使用 head, *tail = C 来提取集合的头部和尾部，如下所示：

```
>>> C= [1,2,3,4,5]
>>> head, *tail = C
>>> head
1
>>> tail
[2, 3, 4, 5]
```

下面用 dict[key(head)].append(head)将头部元素放入结果字典中，然后用 groupby (tail, key)方法处理剩余数据。

如下所示创建函数：

```
from typing import Callable, Sequence, Dict, List, TypeVar
S_ = TypeVar("S_")
K_ = TypeVar("K_")
def group_by(
        key: Callable[[S_], K_],
        data: Sequence[S_]
    ) -> Dict[K_, List[S_]]:

    def group_into(
            key: Callable[[S_], K_],
            collection: Sequence[S_],
            dictionary: Dict[K_, List[S_]]
        ) -> Dict[K_, List[S_]]:
        if len(collection) == 0:
            return dictionary
        head, *tail = collection
        dictionary[key(head)].append(head)
        return group_into(key, tail, dictionary)

    return group_into(key, data, defaultdict(list))
```

内部函数 `group_into()`实现了核心的递归定义部分，当集合 `collection` 为空时返回输入的字典参数。非空集合则分成头部和尾部，头部用于更新字典，尾部包含所有剩余元素，以递归方式更新字典。

类型标示区分了数据源类型 `S_`和键类型 `K_`。作为 `key` 参数的函数必须接收 `S_`类型的参数，返回 `K_`类型的值。前面很多例子都包含了从 `Leg` 对象中取出距离的函数。套用 `Callable[[S_], K_]`，数据源类型 `S_`是 `Leg`，键类型 `K_`是 `float`。

这里不能用 Python 的默认参数将上面的函数简化为简单函数，例如下面的实现方法不可取：

```
def group_by(key, data, dictionary=defaultdict(list)):
```

这样写的话，对 `group_by()`函数的所有调用都将使用同一个 `defaultdict(list)`对象。Python 只创建一次默认值，用可变对象做默认值，其行为往往不符合开发者的预期。这里选用嵌套函数作为实现方案，而没有使用不可变的默认参数（例如 `None`）结合复杂的逻辑判断来实现。用外层包装函数初始化内嵌函数的参数。

下面对距离数据进行分组：

```
binned_distance = lambda leg: 5 * (leg[2] // 5)
by_distance = group_by(binned_distance, trip)
```

首先定义了一个简单且可复用的匿名函数，将距离数据按照 5 海里进行分箱，然后利用该匿名函数对数据进行分组。

检查分箱后的数据，如下所示：

```
import pprint
for distance in sorted(by_distance):
    print(distance)
    pprint.pprint(by_distance[distance])
```

输出结果如下：

```
0.0
[((35.505665, -76.653664), (35.508335, -76.654999), 0.1731),
 ((35.028175, -76.682495), (35.031334, -76.682663), 0.1898),
 ((25.4095, -77.910164), (25.425833, -77.832664), 4.3155),
 ((25.0765, -77.308167), (25.080334, -77.334), 1.4235)]
5.0
[((38.845501, -76.537331), (38.992832, -76.451332), 9.7151),
 ((34.972332, -76.585167), (35.028175, -76.682495), 5.8441),
 ((30.717167, -81.552498), (30.766333, -81.471832), 5.103),
 ((25.471333, -78.408165), (25.504833, -78.232834), 9.7128),
 ((23.9555, -76.31633), (24.099667, -76.401833), 9.844)] ...
125.0
[((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)]
```


或者用迭代方法实现 partition()函数，如下所示：

```
from typing import Callable, Dict, List, TypeVar
S_ = TypeVar("S_")
K_ = TypeVar("K_")
def partition(
        key: Callable[[S_], K_],
        data: Iterable[S_]
    ) -> Dict[K_, List[S_]]:
    dictionary: Dict[K_, List[S_]] = defaultdict(list)
    for head in data:
        dictionary[key(head)].append(head)
    return dictionary
```

进行尾调用优化时，命令式实现的核心部分与递归定义是一致的。前面已经分析了这行代码，保证改写后的输出与之前一样。其余部分作为解决 Python 递归限制的常规编程实践，与之前的尾调用优化实现是一致的。

类型标示区分了数据源类型 S_和键类型 K_。请注意，defaultdict(list)的返回值要用 Dict[K_, List[S_]]标示协助 mypy 工具确认代码正常运行，否则会返回错误信息：error: Need type annotation for variable。defaultdict 可以包含任何类型组合，如果没有类型标示，将无法确认是否使用了正确的类型。

也可以把这里的类型标示写成注释的形式，如下所示：

```
dictionary = defaultdict(list) # type: Dict[K_, List[S_]]
```

低版本的 pylint 工具要求必须这样写，因此推荐使用 1.8 及之后的版本。

6.2.4　编写更通用的 group-by 归约

将原始数据分区后，就可以对每个分区中的数据执行多种归约操作了，例如提取出每个路径段中最靠北的起点。

首先引入如下辅助函数来拆解元组：

```
start = lambda s, e, d: s
end = lambda s, e, d: e
dist = lambda s, e, d: d
latitude = lambda lat, lon: lat
longitude = lambda lat, lon: lon
```

每个辅助函数的输入值是一个元组，通过*运算符将元组中的每个元素拆解为匿名函数的多个输入参数。当把元组的元素拆解为 s、e 和 p 这 3 个参数后，就可以方便地按名称得到返回值了。这种方法比 tuple_arg[2]方式更明了。

如下所示使用这些辅助参数：

```
>>> point = ((35.505665, -76.653664), (35.508335, -76.654999), 0.1731)
>>> start(*point)
(35.505665, -76.653664)
>>> end(*point)
(35.508335, -76.654999)
>>> dist(*point)
0.1731
>>> latitude(*start(*point))
35.505665
```

输入数据是嵌套三元组，按照下标依次为起点、终点和距离。利用上面定义的辅助函数，就能抽取不同的元素。

基于这些辅助函数，提取每个箱子中起点最靠北的路径段的实现如下所示：

```
for distance in sorted(by_distance):
    print(
        distance,
        max(by_distance[distance],
            key=lambda pt: latitude(*start(*pt)))
    )
```

首先按距离将各个路径段分组，用这些路径段作为 max()函数的输入，这个函数的 key 参数是一个从路径段中提取出起点纬度的匿名函数。

返回结果是每个箱子中起点最靠北的路径段列表，如下所示：

```
0.0 ((35.505665, -76.653664), (35.508335, -76.654999), 0.1731)
5.0 ((38.845501, -76.537331), (38.992832, -76.451332), 9.7151)
10.0 ((36.444168, -76.3265), (36.297501, -76.217834), 10.2537)
...
125.0 ((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)
```

6.2.5 编写高阶归约

下面介绍相对复杂的高阶归约算法。最简单的归约形式是从集合数据中算出一个简单的标量值。Python 内置了归约函数，包括 any()、all()、max()、min()、sum()和 len()。

第 4 章讲过，可以使用简单的归约函数计算很多统计值，如下所示：

```
def s0(data: Sequence) -> float:
    return sum(1 for x in data) # or len(data)
def s1(data: Sequence) -> float:
    return sum(x for x in data) # or sum(data)
def s2(data: Sequence) -> float:
    return sum(x * x for x in data)
```

使用几个简单的函数，就可以定义平均值、标准差、归一值、修正值甚至最小线性回归函数了。

上面最后一个简单归约 s2()展示了如何基于已有归约函数创建高阶函数，可如下所示改写实现方法：


```
from typing import Callable, Iterable, Any
def sum_f(
        function: Callable[[Any], float],
        data: Iterable) -> float:
    return sum(function(x) for x in data)
```

其中有一个参数用于定义转换数据方法，计算结果是转换后的数值之和。

对于该函数有 3 种用法，可得到不同的和，如下所示：

```
N = sum_f(lambda x: 1, data) # x ** 0
S = sum_f(lambda x: x, data) # x ** 1
S2 = sum_f(lambda x: x * x, data) # x ** 2
```

使用简单的匿名函数，计算 $\sum_{x \in X} x^0 = \sum_{x \in X} 1$，即序列长度，$\sum_{x \in X} x^1 = \sum_{x \in X} x$，即序列和，以及 $\sum_{x \in X} x^2$，序列元素平方和，然后就能计算出标准差了。

在此基础上，添加过滤器来扩展该算法，使其能去除序列中无效或者不希望保留的元素。去除无效数据如下所示：

```
def sum_filter_f(
        filter_f: Callable,
        function: Callable, data: Iterable) -> Iterator:
    return sum(function(x) for x in data if filter_f(x))
```

执行以下命令可过滤掉 None 值并计算序列和：

```
count_ = lambda x: 1
sum_ = lambda x: x
valid = lambda x: x is not None
N = sum_filter_f(valid, count_, data)
```

这段代码演示了 `sum_filter_f()`函数是如何使用两个匿名函数的。其中过滤器用于去除序列中的 `None` 值，所以将其命名为 `valid` 以体现其功能。`function` 参数是执行 `count` 方法或者 `sum` 方法的匿名函数，同理，计算平方和也很方便。

需要说明的是，该函数与其他高阶函数类似，返回的是一个函数，而不是一个值。这是高阶函数的一个核心特征，并且用 Python 易于实现。

6.2.6　编写文件解析器

可以将文件解析看作归约。许多语言使用了双层定义：语言的底层标记，以及建于其上的高级结构。以 XML 文件为例，标签、标签名称以及属性名称构成了底层语法，XML 描述的整体结构构成了高级语法。

底层的文本扫描实际上是对单个字符的归约，并把它们分组为标记，用 Python 的生成器函数设计模式可以轻松实现。常规的实现方法如下：

```
def lexical_scan(some_source):
    for char in some_source:
        if some pattern completed: yield token
        else: accumulate token
```

可以使用底层文件解析器来完成这部分工作，使用 CSV、JSON 和 XML 库处理细节，我们会基于这些库开发高级解析器。

沿用双层设计模式，底层解析器生成原始数据的正式数据结构，即由文本元组组成的迭代器，兼容多种格式的数据文件。高级解析器则负责生成可供应用程序使用的对象，可以是数字元组、命名元组或者其他类型的 Python 不可变对象。

第 4 章用到了一种底层解析器，其输入是 KML 文件，KML 文件是 XML 表达地理位置信息的一种格式。解析器的核心功能如下：

```
def comma_split(text: str) -> List[str]:
    return text.split(",")
def row_iter_kml(file_obj: TextIO) -> Iterator[List[str]]:
    ns_map = {
        "ns0": "http://www.opengis.net/kml/2.2",
        "ns1": "http://www.google.com/kml/ext/2.2"}
    xpath = (
        "./ns0:Document/ns0:Folder/"
        "ns0:Placemark/ns0:Point/ns0:coordinates")
    doc = XML.parse(file_obj)
    return (
        comma_split(cast(str, coordinates.text))
        for coordinates in doc.findall(xpath, ns_map)
    )
```

`row_iter_kml()`函数解析 XML 文件的方法是通过 `doc.findall()`函数遍历文档中所有

的<ns0:coordinates>标签，然后通过 comma_split()函数将标签中的文本转换为三元组。

cast()函数的作用是为 mypy 标示出 coordinates.text 是字符串类型值，text 属性的默认类型是 Union[str, bytes]，但这里只有字符串这种情况。cast()不执行任何运行时操作，仅用作 mypy 的类型标示。

该解析器可以处理标准化的 XML 结构。输入文档必须满足数据库设计中第一范式的要求，即每个属性必须满足原子化和单一值的要求，XML 数据的每一行拥有相同的列，每列中元素的类型一致。数据的值并非完全原子化的，需要用逗号分隔为经度、纬度和海拔高度的原子字符串。但文本类型完全一致，符合第一范式。

将大量数据（包括 XML 标签、属性以及其他标记符号）归约为相对少量的信息，即浮点数类型的经度和纬度，所以解析器实际上进行的是某种归约计算。

现在需要将元组中的字符串转换为浮点数，去掉海拔高度，调整经纬度顺序，从而得到符合应用程序要求的元组格式，具体变换过程如下所示：

```
def pick_lat_lon(
        lon: Any, lat: Any, alt: Any) -> Tuple[Any, Any]:
    return lat, lon

def float_lat_lon(
        row_iter: Iterator[Tuple[str, ...]]
    ) -> Iterator[Tuple[float, ...]]:
    return (
        tuple(
            map(float, pick_lat_lon(*row))
        )
        for row in row_iter
    )
```

关键所在的 float_lat_lon()是高阶函数，返回生成器表达式。其中生成器使用 map()函数将 float()函数应用于 pick_lat_lon()函数的返回结果，参数*row 将行数据元组的每个元素取出作为 pick_lat_lon()函数的各个参数，所以只有每行为三元组时才可用。pick_lat_lon()函数再对输入值进行筛选和重新排序，返回符合要求的元组。

如下所示使用解析器：

```
name = "file:./Winter%202012-2013.kml"
with urllib.request.urlopen(name) as source:
    trip = tuple(float_lat_lon(row_iter_kml(source)))
```

返回结果是从原来 KML 文件中转换来的路径上的一系列路径点，每个点采用嵌套元组的形式。使用底层解析器从原始数据中抽取行文本，再使用高级解析器将文本转换为浮点数元组，这里没有执行任何验证。

1. 解析 CSV 文件

第 3 章介绍了另一个解析非标准 CSV 文件的例子。当时为了去掉文件头，使用了一个简单的函数提取文件头，然后返回了包含剩余行数据的迭代器。

原始数据如下所示：

```
Anscombe's quartet
I   II    III     IV
x   y    x   y    x    y   x   y
10.0     8.04     10.0     9.14    10.0   7.46     8.0     6.58
8.0     6.95     8.0    8.14    8.0   6.77   8.0      5.76
...
5.0     5.68     5.0    4.74    5.0   5.73   8.0      6.89
```

数据之间以 Tab 字符分隔，文件前 3 行是需要去掉的文件头部分。

下面是 CSV 解析器的另一个版本，我们通过 3 个函数来实现。第一个函数 `row_iter()`返回包含 Tab 分隔符的迭代器，如下所示：

```
def row_iter_csv(source: TextIO):
    rdr= csv.reader(source, delimiter="\t")
    return rdr
```

只是简单地包装了 CSV 文件解析器，之前实现过的 XML 和普通文本解析器并不包含这一部分。解析器处理标准格式数据的常规方法是创建基于行元组的迭代器。

得到行元组后，就可以保留包含可用数据的行，去掉包含其他元数据的行，例如标题和列名称。这里需要引入一个用于解析的辅助函数，以及用于验证行数据的过滤函数。

转换函数如下所示：

```
from typing import Optional, Text
def float_none(data: Optional[Text]) -> Optional[float]:
    try:
        data_f = float(data)
        return data_f
    except ValueError:
        return None
```

该函数负责将格式正确的字符串转换为浮点数，将无效数据转换为 `None`。类型标示 `Optional[Text]`和 `Optional[float]`表示对应值是符合类型定义的普通值或者 `None` 值。

下面把该函数嵌入到一个映射中，从而将整行数据转换为浮点数或者 `None` 值，该匿名函数如下所示：

```
from typing import Callable, List, Optional
R_Text = List[Optional[Text]]
R_Float = List[Optional[float]]
```

```
float_row: Callable[[R_Text], R_Float] \
    = lambda row: list(map(float_none, row))
```

两个类型标示显式定义了 float_row。R_Text 定义了行数据的文本版本是个包含了文本值和 None 值的列表。R_Float 定义了行数据的实数版本。

下面是一个行级验证器，使用 all() 函数确保所有值都是浮点数（或者说没有 None 值）：

```
all_numeric: Callable[[R_Float], bool] \
    = lambda row: all(row) and len(row) == 8
```

该匿名函数属于归约函数，将一行实数归约为一个布尔值。如果这些数据都不为“假”（既不是 None 值，也不是零值），并且个数正好是 8，则返回布尔“真”值。

上面这个简化版的 all_numeric 函数不区分零值和 None 值。更准确的测试条件应该类似于 not any(item is None for item in row)，具体实现留给读者作为练习。

6

该函数设计的核心目标是创建基于行的元素，并将其整合到完整算法中以解析输入文件。基本函数迭代处理文本元组，联合起来对结果数据进行转换及验证。对于满足第一范式（所有行相同）或者可以通过简单方法排除异常数据行的情况，该函数表现不错。

但大多数解析工作不会这么简单。有些文件的头部或者尾部包含重要数据，虽然与其他部分的格式不同，却不能简单地丢弃，需要用更复杂的解析器处理这类非标准文件。

2. 解析带文件头的普通文本文件

第 3 章有个没有经过解析处理的 Crayola.GPL 文件，如下所示：

```
GIMP Palette
Name: Crayola
Columns: 16
#
239 222 205   Almond
205 149 117   Antique Brass
```

可以使用正则表达解析文本文件。首先用一个过滤器读取并解析文件头，然后返回一个包含数据行的可迭代序列。这个复杂的二步解析器完全是基于由两部分（文件头和尾）组成的文件结构定义的。

如下底层解析器可以处理包含 4 行文件头和大量尾部数据行：

```
Head_Body = Tuple[Tuple[str, str], Iterator[List[str]]]
def row_iter_gpl(file_obj: TextIO) -> Head_Body:
    header_pat = re.compile(
        r"GIMP Palette\nName:\s*(.*?)\nColumns:\s*(.*?)\n#\n",
        re.M)
```

```
    def read_head(
            file_obj: TextIO
        ) -> Tuple[Tuple[str, str], TextIO]:
        match = header_pat.match(
            "".join(file_obj.readline() for _ in range(4))
        )
        return (match.group(1), match.group(2)), file_obj

    def read_tail(
            headers: Tuple[str, str],
            file_obj: TextIO) -> Head_Body:
        return (
            headers,
            (next_line.split() for next_line in file_obj)
        )

    return read_tail(*read_head(file_obj))
```

Head_Body 类型定义了行迭代器的整体实现目标，返回结果是个二元组。其中第一个元素也是一个二元组，包含从头部提取的信息，第二个元素是一个包含颜色信息的迭代器。下面的函数有两处用到了这个 Head_Body 类型定义。

header_pat 是一个用于解析前 4 行文件头的正则表达式，表达式中的两个括号用于提取头部的名称和列数信息。

接下来定义了两个内部函数来解析文件的不同部分。read_head()函数解析文件头，具体过程是：首先读取文件的前 4 行，将其合并为一个字符串，然后用 head_pat 正则表达式进行解析，返回结果包含了文件头中的两部分数据，以及处理剩余行的迭代器。

一个函数返回迭代器作为另一个函数的输入参数，基于在函数间传递状态对象的设计思想，但它只能算作一个简化实现，因为 read_tail()函数的输入参数正是 read_head()函数的输出。

read_tail()函数解析文件的剩余行，按照 GPL 文件格式的定义来解析，基于空格符分割字符串。



更多信息，可访问 https://code.google.com/p/grafx2/issues/detail?id=518。

当把文件中的每行文本解析为一系列字符串元组后，就可以用高阶函数进行后续处理了，包括转换和验证（如若需要）。

高阶解析函数示例如下：

```
from typing import NamedTuple
class Color(NamedTuple):
    red: int
    blue: int
```

```
        green: int
        name: str

    def color_palette(
            headers: Tuple[str, str],
            row_iter: Iterator[List[str]]
        ) -> Tuple[str, str, Tuple[Color, ...]]:
        name, columns = headers
        colors = tuple(
            Color(int(r), int(g), int(b), " ".join(name))
            for r, g, b, *name in row_iter)
        return name, columns, colors
```

该函数所需的头部数据与迭代器由底层函数 `row_iter_gpl()`提供。使用多重赋值将颜色数据与剩余的名称分开，分别赋给 4 个变量 `r`、`g`、`b` 和 `name`。使用`*name` 形式的参数以确保所有剩余数据都会以元组的形式赋给保存名字的变量。最后`" ".join(name)`将这些单词用空格连接起来组成一个完整的字符串。

如下所示使用该双层解析器：

```
with open("crayola.gpl") as source:
    name, cols, colors = color_palette(
        *row_iter_gpl(source)
    )
print(name, cols, colors)
```

对底层函数的处理结果应用高阶函数，最终的返回结果是文件头数据和一系列 `Color` 对象组成的元组。

6.3 小结

本章讨论了函数式编程的两个重要主题。首先详细介绍了递归技术，许多函数式编程语言编译器会将递归函数的尾调用转换为循环，但在 Python 中，我们需要手动将纯函数的递归转换为显式 `for` 循环，以实现尾调用优化的效果。

然后介绍了归约函数，包括 `sum()`、`count()`、`max()`和 `min()`，以及与集合相关的 `collcetions.Counter()`和 `groupby()`归约函数。

由于文本解析是将标记序列（或者字符串序列）转换为带有复杂属性的高阶集合，所以本质上也属于归约。常用的设计模式是将整个解析过程分为底层解析函数和高级解析函数，前者将原始字符串转换为元组，后者在此基础上创建符合应用格式要求的对象。

下一章将介绍与命名元组以及其他不可变数据结构相关的处理技术，这样应用程序就不再需要有状态的对象了。有状态对象不是纯粹函数式的，但借鉴类继承的思想，我们可以将相关函数定义组织在一起。

第 7 章 元组处理技术

前面的例子有的是标量函数，有的是用基本元组创建一些相对简单的结构。在 Python 中，经常用不可变的命名元组（`typing.NamedTuple`）创建复杂的数据结构。

面向对象编程的一个优点是可以渐进地创建复杂的数据结构。有时对象用作保存函数计算结果的缓存，这种情况很符合函数式设计模式的思想。有时对象的属性方法中定义了基于对象属性获得数据的复杂计算过程，同样可以方便地转换为函数式设计。

而在某些情况下，对象类定义了如何通过状态变化创建复杂对象。我们将介绍几种方法降低有状态对象导致的复杂度，首先识别出有状态的类定义，然后引入元属性实现方法按序调用，例如“如果调用 `X.p()` 先于 `X.q()`，则返回结果为 `undefined`”这类语句不属于语言形式，而是类的元属性。有些有状态类中包含显式断言，或者用于保证方法调用顺序的错误检查所产生的开销，如果不使用有状态类，就可以避免这些额外的开销。

本章将介绍以下内容。

- 如何创建并使用命名元组。
- 用不可变的命名元组代替有状态的对象类的方法。
- 不依赖多态类而编写抽象函数的技术。当然，可以基于 `Callable` 类创建多态类继承结构，但在某些情况下，使用函数式设计可以避免这些不必要的开销。

7.1 使用元组收集数据

第 3 章介绍了两种常用的处理元组的技术，并简单介绍了处理复杂结构的第三种方法。根据具体场景，可选用下面任一种：

- 使用匿名函数，通过下标取值；
- 使用匿名函数，通过带星号的参数将参数名称映射到下标来取值；
- 使用命名元组，通过属性名称或者下标取值。

第 4 章的旅行数据就是一个复杂结构的例子，原始数据是由位置点组成的普通时间序列。为了计算距离，我们将上述数据转换成了一系列路径段，每个元素是由起点、终点和距离组成的三元组。

每个路径段形如下面的三元组：

```
first_leg = (
    (37.549016, -76.330295),
    (37.840832, -76.273834),
    17.7246)
```

第一项和第二项分别代表起点和终点，第三项代表两点间的距离。这是 Chesapeake Bay 两地间的短途旅行。

嵌套元组可读性不佳，比如 `first_leg[0][0]`很不容易理解。

除了元组之外的其他三种取值方法，第一种方法是定义一个简单的选择函数，通过下标位置取值。

```
start = lambda leg: leg[0]
end = lambda leg: leg[1]
distance = lambda leg: leg[2]
latitude = lambda pt: pt[0]
longitude = lambda pt: pt[1]
```


基于上面的函数，可以用 `latitude(start(first_leg))`取特定的值，如下所示：

```
>>> latitude(start(first_leg))
29.050501
```

上面的定义没有提供关于返回结果的数据类型的信息，可以按照命名规则改进定义。为函数名称增加后缀的定义如下所示：

```
start_point = lambda leg: leg[0]
distance_nm = lambda leg: leg[2]
latitude_value = lambda point: point[0]
```

运用得当的话这种方法很有用，也可以用复杂的匈牙利命名法设置前缀（或者后缀）为各个变量命名。

为匿名函数添加类型标示有点画蛇添足，可做如下尝试：

```
>>> from typing import Tuple, Callable
>>> Point = Tuple[float, float]
>>> Leg = Tuple[Point, Point, float]
>>> start: Callable[[Leg], Point] = lambda leg: leg[0]
```

类型标示作为赋值语句的一部分，告知 mypy`start` 对象是可调用函数，接收类型为 `Leg` 的参数，返回结果的类型为 `Point`。

第二种方法是用带星号的参数隐藏下标位置。使用了*标记的函数示例如下：

```
start = lambda start, end, distance: start
end = lambda start, end, distance: end
distance = lambda start, end, distance: distance
latitude = lambda lat, lon: lat
longitude = lambda lat, lon: lon
```

基于以上函数，可以用 latitude(*start(*first_leg))从原始数据中取值，如下所示：

```
>>> latitude(*start(*first_leg))
29.050501
```

这种方法的优点是可读性较好，位置和名称之间的关系由一组参数名称定义。虽然在元组参数前面加上星号使得函数看上去有点别扭，但星号运算符是必要的，它抽取元组中各个元素并将其分别赋给函数的每个参数。

这种方式更为函数式，但选择单个属性的语法容易混淆，Python 提供了一种面向对象风格的替代方案：命名元组。

7.2　使用命名元组收集数据

将数据收集到复杂数据结构中的第三种方法是使用命名元组，基本的思路是创建一种带名称属性的元组，有如下两种实现方法：

- 使用 collections 模块的 namedtuple 函数；
- typing 模块的 NamedTuple 基类，这种方式支持类型标示，本书基本上只用这种方法。

前面章节的例子中命名元组类定义如下：

```
from typing import NamedTuple

class Point(NamedTuple):
    latitude: float
    longitude: float

class Leg(NamedTuple):
    start: Point
    end: Point
    distance: float
```

这样原来的匿名元组的每个属性都有各自的类型标示了，示例如下：

```
>>> first_leg = Leg(
... Point(29.050501, -80.651169),
... Point(27.186001, -80.139503),
... 115.1751)
>>> first_leg.start.latitude
29.050501
```

first_leg 对象的类型是 NamedTuple 类的子类 Leg，这个类包含了另外两个命名元组和一个实数。用 first_leg.start.latitude 就可以获取元组结构的指定数据，这种从前缀函数方式到后缀属性方式的变化，既可以看作一种强调，也可以看作某种让人困惑的语法变化。

用 Leg()和 Point()函数代替 tuple()函数很重要，它改变了构造数据结构的流程，并且为 mypy 提供了明确的命名结构来验证类型标示是否正确。

从源数据种创建点对的方法如下：

```
from typing import Tuple, Iterator, List

def float_lat_lon_tuple(
        row_iter: Iterator[List[str]]
    ) -> Iterator[Tuple]:
    return (
        tuple(*map(float, pick_lat_lon(*row)))
        for row in row_iter
    )
```

它处理一个迭代器（例如 CSV 读取器或者 KML 读取器）给出的一系列字符串，pick_lat_lon()函数从行字符串中取出两个值，map()函数将 float()函数应用于这些取出的值，返回结果是普通的元组。

为了创建 Point 对象，要对上面的函数做如下更改：

```
def float_lat_lon(
        row_iter: Iterator[List[str]]
    ) -> Iterator[Point]:
    return (
        Point(*map(float, pick_lat_lon(*row)))
        for row in row_iter
    )
```

用Point()构造函数代替了tuple()函数，返回的数据类型也相应地变成了Iterator[Point]，表明函数构造的是实数坐标组成的点对象，而不是匿名元组。

可以采用类似的方法构建旅行数据的完整 Leg 对象，如下所示：

```
from typing import cast, TextIO, Tuple, Iterator, List
from Chapter_6.ch06_ex3 import row_iter_kml
from Chapter_4.ch04_ex1 import legs, haversine

source = "file:./Winter%202012-2013.kml"
def get_trip(url: str=source) -> List[Leg]:
    with urllib.request.urlopen(url) as source:
        path_iter = float_lat_lon(row_iter_kml(
            cast(TextIO, source)
        ))
        pair_iter = legs(path_iter)
```

7

```
        trip_iter = (
            Leg(start, end, round(haversine(start, end), 4))
            for start, end in pair_iter
        )
        trip = list(trip_iter)
    return trip
```

整个处理过程由一系列生成器表达式组成。path_iter 对象使用两个生成器函数 row_iter_kml()和 float_lat_lon()从 KML 文件中读取每行文本,提取数据并将其转换为 Point 对象。pair_iter 对象使用 legs()生成器函数创建代表每段路径起点和终点的 Point 对象二元组。

trip_iter 将 Point 二元组转换为最终的 Leg 对象，list()函数再将这些对象变为一个 Leg 序列，第 4 章中定义的 haversine()函数负责计算起点和终点间的距离。

cast()函数用于通知 mypy 工具 source 对象是 TextIO 类的一个实例。cast()函数是一个类型标示，不包含运行时操作。由于 urlopen()返回值的类型是 Union[HTTPResponse, addinfourl]，所以 cast()函数是必需的。addinfourl 对象的类型是 BinaryIO。csv.reader()要求输入参数的类型为 List[str]，需要 urlopen()提供文本而非字节。对于简单的 CSV 文件来说，字节和 UTF-8 编码的文本的差别不大，使用 cast()即可。

为了能正确处理字节，需要使用 codecs 模块将字节转换为正确编码的文本，如下所示：

```
cast(TextIO, codecs.getreader('utf-8')(cast(BinaryIO, source)))
```

最内层的 cast()函数用于通知 mypy 工具 source 的类型是 BinaryIO。codecs.getreader()创建能正确处理 UTF-8 编码的读取器，这个类的实例基于 source 对象创建文件读取器。

返回结果是一个 StreamReader 对象，最外面的 cast()函数通知 mypy 工具将 StreamReader 作为 TextIO 的实例来处理。codecs.getreader()创建的读取器是将由字节组成的文件解码为格式正确的文本的关键。其他的类型变换都是提供给 mypy 工具的类型标示。

trip 对象是由 Leg 实例组成的序列，打印结果如下：

```
(Leg(start=Point(latitude=37.549016, longitude=
 -76.330295), end=Point(latitude=37.840832, longitude=
 -76.273834), distance=17.7246),
 Leg(start=Point(latitude=37.840832, longitude=-76.273834),
 end=Point(latitude=38.331501, longitude=-76.459503),
 distance=30.7382),
 ...
 Leg(start=Point(latitude=38.330166, longitude=-76.458504),
 end=Point(latitude=38.976334, longitude=-76.473503),
 distance=38.8019))
```



需要说明的是，原来的 haversine()函数处理的是简单的元组，这里用它来处理命名元组，由于输入参数的顺序没变，所以从元组到命名元组的变化不会影响 Python 的处理过程。

大多数情况下，使用 NamedTuple 有助于提高程序的可读性，并能把前缀式函数风格变成后缀式对象风格。

7.3 使用函数构造器创建命名元组

创建命名元组实例有三种方法，具体选择取决于创建实例时有多少附加信息可用。

前面的例子展示了三种方法中的两种，接下来重点介绍设计时要考虑的因素，包括以下选项。

- 根据位置对参数赋值，当需要对一个或者多个表达式求值时，可用这种方法。例如下面的代码中将 haversine()函数应用于 start 和 end 来创建 Leg 对象：Leg(start, end, round(haversine(start, end),4))
- 使用星号参数，按照在元组中的位置给参数赋值。这种方法适用于从其他可迭代对象或者已有元组中获取参数值。通过 map()函数把 float()函数应用于经纬度数据时，用到了该方法：Point(*map(float, pick_lat_lon(*row)))
- 使用显式参数名称赋值，前面的例子没有用过这种方法，优点是可以使参数赋值关系更加明确：Point(longitude=float(row[0]), latitude=float(row[1]))

使用多种方式创建命名元组实例有助于灵活地转换数据。可以通过不同的方式强调那些有助于阅读和理解应用代码的数据结构特征，有时需要突出作为下标的 0 或者 1，有时则需要强调起点、终点和距离的顺序。

7.4 使用多种元组结构代替状态类

前面的例子使用了**打包–拆包**设计模式处理不可变元组和命名元组，其核心思想是用不可变对象包含其他不可变对象，来代替面向对象范式中的可变实例变量。

斯皮尔曼等级相关系数是一种用于表征两组变量相关度的统计量。比较两组变量的等级，由于变量值可能在尺度上有差异，所以它不比较具体的值，而比较相对顺序。有关该算法的更多信息，可参考维基百科。

计算斯皮尔曼等级相关系数时需要给每个样本赋一个等级值，使用 enumerate(sorted()) 可以实现。对给定的可能存在相关性的两组数据，分别将每组数据转换为一系列等级值，然后计算相关度。

这里使用“打包–拆包”设计模式实现算法，首先用等级值打包数据，便于后续计算相关系数。

第 3 章介绍了如何解析数据集。从数据集中提取 4 份样本的做法如下：

```
>>> from Chapter_3.ch03_ex5 import (
...     series, head_map_filter, row_iter)
>>> with open("Anscombe.txt") as source:
...     data = list(head_map_filter(row_iter(source)))
```

得到的数据集每行包含 4 个序列，series()函数从所有行中提取指定序列，返回结果是个二元组，如果是命名元组会更好。

代表每对数据的命名元组如下：

```
from typing import NamedTuple

class Pair(NamedTuple):
    x: float
    y: float
```

下面引入一个变换函数，将匿名元组转换为命名元组。

```
from typing import Callable, List, Tuple, Iterable
RawPairIter = Iterable[Tuple[float, float]]

pairs: Callable[[RawPairIter], List[Pair]] \
    = lambda source: list(Pair(*row) for row in source)
```

RawPairIter 类型代表 series()函数返回的中间输出结果：一个输出二元组的可迭代序列。pairs 匿名函数接收一个可迭代序列，返回一个由 Pair 命名元组组成的列表。

使用 pairs()函数和 series()函数从源数据中抽取数据对，如下所示：

```
>>> series_I = pairs(series(0, data))
>>> series_II = pairs(series(1, data))
>>> series_III = pairs(series(2, data))
>>> series_IV = pairs(series(3, data))
```

每个序列是一个由 Pair 对象组成的列表，每个 Pair 对象包含 x 和 y 两个属性，数据如下所示：

```
[Pair(x=10.0, y=8.04),
 Pair(x=8.0, y=6.95),
 ...,
 Pair(x=5.0, y=5.68)]
```

为了便于计算等级，需要构造一个包含等级值和原始数据对的组合对象，该二元组的类型定义如下所示：

```
from typing import Tuple
RankedPair = Tuple[int, Pair]
```

Pair 是前面定义的命名元组，RankedPair 是包含一个整数和一个 Pair 对象的二元组的类型别名。

如下所示的生成器函数将包含 Pair 的可迭代集合转换为 RankedPair：

```
from typing import Iterable, Iterator
def rank_y(pairs: Iterable[Pair]) -> Iterator[RankedPair]:
    return enumerate(sorted(pairs, key=lambda p: p.y))
```

对 RankedPair 对象应用 enumerate() 函数创建迭代器。按 Pair 对象的 y 值排序。将每个 Pair 对象和一个等级值包装成一个二元组。

更复杂的实现如下：

```
Rank2Pair = Tuple[int, RankedPair]
def rank_x(
        ranked_pairs: Iterable[RankedPair]
    ) -> Iterator[Rank2Pair]:
    return enumerate(
        sorted(ranked_pairs, key=lambda rank: rank[1].x)
)
```


将 RankedPair 对象包装进 Rand2Pair 对象中。第二次包装生成了一个包含二元组的二元组，这个复杂的数据结构表明类型别名能为被处理的数据提供有效的类型提示。

y_rank = list(rank_y(series_I)) 的运行结果如下：

```
[(0, Pair(x=8.0, y=5.25)),
 (1, Pair(x=8.0, y=5.56)),
 ...,
 (10, Pair(x=19.0, y=12.5))
]
```

为了计算相关度，需要使用 rank_x() 函数和 rank_y() 函数，xy_rank = list(rank_x(y_rank)) 的值是由深度嵌套对象组成的列表，如下所示：

```
[(0, (0, Pair(x=4.0, y=4.26))),
 (1, (2, Pair(x=5.0, y=5.68))),
 ...,
 (10, (9, Pair(x=14.0, y=9.96)))
]
```

这样就可以基于 x 和 y 的等级值，而不是原始 Pair 对象计算等级序列的相关度了。

要提取两个等级值，需要两个复杂表达式。对于数据集中每个标记了等级值的样本 r，需要比较 r[0] 和 r[1][0]，这是与前面包装过程相对应的拆包过程。有时也将这类函数称为**选择器函数**，因为它们从复杂的数据结构中选择数据项。

为了避免对 r[0] 和 r[1][0] 的复杂引用，可以创建如下所示的选择器函数：

```
x_rank = lambda ranked: ranked[0]
y_rank = lambda ranked: ranked[1][0]
raw = lambda ranked: ranked[1][1]
```

这样就可以通过 x_rank_(r) 和 y_rank_(r) 来计算相关度了，引用表达式比原来的版本易读。

总的处理策略包含两部分操作：包装和拆包，rank_x() 函数和 rank_y() 函数包装 Pair 对象，用等级值和原始数据创建元组。通过创建复杂度逐渐增加的数据结构，避免了使用有状态的类定义。

为什么要创建深层嵌套元组呢？原因很简单：惰性求值。对元组进行拆包，生成新的扁平元组很花时间，包装已有元组则简单多了。使用扁平数据结构能大幅降低后续处理的复杂度，下面对已有的处理逻辑做如下改进。

- 平铺数据结构，rank_x() 函数和 rank_y() 函数的类型标示显示了其复杂度，一个在 Tuple[int, Pair] 上迭代，另一个在 Tuple[int, RankedPair] 上迭代。
- enumerate() 函数不能正确计算有多个相同值序列的等级值。如果样本中有两个值大小相同，相应的等级值应相同。值的大小应该是这些相同值位置的平均数，例如序列 [0.8, 1.2, 1.2, 2.3, 18] 的等级值是 1, 2.5, 2.5, 4, 5，处在第 2 位和第 3 位的两个相同值的等级值是它们位置的平均值：2.5。

下面通过编写更智能的等级计算函数完成上述优化。

7.4.1　赋等级值

下面将等级排序问题分为两部分来处理。首先创建一个通用的高阶函数给 Pair 对象的 x 或者 y 属性赋等级值，接着用它包装 Pair 对象，将 x 和 y 属性的等级值纳入其中，从而避免深层嵌套结构。

为数据集中的每个样本赋等级值的函数如下所示：

```
from typing import Callable, Tuple, List, TypeVar, cast, Dict
from typing import Dict, Iterable, Iterator
from collections import defaultdict
D_ = TypeVar("D_")
K_ = TypeVar("K_")
def rank(
        data: Iterable[D_],
        key: Callable[[D_], K_]=lambda obj: cast(K_, obj)
    ) -> Iterator[Tuple[float, D_]]:

    def build_duplicates(
            duplicates: Dict[K_, List[D_]],
            data_iter: Iterator[D_],
            key: Callable[[D_], K_]
```

```
        ) -> Dict[K_, List[D_]]:
        for item in data_iter:
            duplicates[key(item)].append(item)
        return duplicates

    def rank_output(
            duplicates: Dict[K_, List[D_]],
            key_iter: Iterator[K_],
            base: int=0
        ) -> Iterator[Tuple[float, D_]]:
        for k in key_iter:
            dups = len(duplicates[k])
            for value in duplicates[k]:
                yield (base + 1 + base + dups) / 2, value
            base += dups

    duplicates = build_duplicates(
        defaultdict(list), iter(data), key)
    return rank_output(duplicates, iter(sorted(duplicates)), 0)
```

该等级排序函数使用两个子函数将值列表转换为包含序列值和源数据的二元组列表。第一步使用 `build_duplicates()`函数创建字典 `duplicates`，字典的键值是由源数据经 `key` 函数转换后的值，键值对应的值是由拥有相同键值的源数据组成的序列。第二步调用 `rank_output()` 函数，在 `duplicates` 的基础上生成二元组序列。

为了厘清数据之间的关系，这里定义了两个类型变量。`D_`类型变量代表源数据的类型，例如 `Leg` 类型或者其他复杂对象。`K_`类型变量代表用于排序的等级值的类型，它可以与源数据类型不同，例如从 `Leg` 命名元组中取出的距离值是实数类型。从源数据到等级值的转换是通过 `key` 参数代表的函数参数实现的，该参数的类型标示是 `Callable[[D_], K_]`。

`build_duplicates()`函数使用有状态的对象构造键值之间的映射关系，具体实现是通过对递归算法进行尾调用优化得到的。首先将内部状态作为 `build_duplicates()`函数的参数暴露出来。递归的基本场景是 `data_iter` 为空，`base` 为 0。对于算法的迭代实现版本，这些变量并不是必需的，但使用它们可以更好地说明递归的工作方式。

与之类似，`rank_output()`函数也可以通过递归生成包含源数据和等级值的二元组，这里将其优化为二重 `for` 循环。为了显式计算出等级值，取左边界 `base + 1` 和右边界 `base + dups` 的平均值，如果 `duplicates` 的某一项只有一个元素，则等级值是`(2 * base + 2) / 2`，所以该公式的普适性较好。

`duplicates` 的类型标示是 `Dict[K_, List[D_]]`，表示键类型是 `K_`，值类型是 `List[D_]`，也就是由源数据组成的列表。这个类型在算法中多次出现，表明好的类型定义能很好地说明类型间的复用关系。

如下所示测试其能否正常工作，第一个例子是对单个值排序，第二个例子是对二元组列表排序，使用匿名函数从每组中取键值。

```
>>> list(rank([0.8, 1.2, 1.2, 2.3, 18]))
[(1.0, 0.8), (2.5, 1.2), (2.5, 1.2), (4.0, 2.3), (5.0, 18)]
>>> data= [(2, 0.8), (3, 1.2), (5, 1.2), (7, 2.3), (11, 18)]
>>> list(rank(data, key=lambda x:x[1]))
[(1.0, (2, 0.8)),
(2.5, (3, 1.2)),
(2.5, (5, 1.2)),
(4.0, (7, 2.3)),
(5.0, (11, 18))]
```

样本数据中包含了两个相同值，等级值的最终计算结果是两个顺序（2 和 3）的中间数 2.5，这是计算两个数据集的斯皮尔曼等级相关系数的常用方法。



rank()函数重排了输入数据以查找重复值。如果要对每一对数据的 x 和 y 值求等级值，就需要对数据进行两次重排。

7.4.2　用包装代替状态变化

可以按照以下两种策略包装数据。

- **并行**：创建两份数据副本，分别对每一份求等级值，然后合并两份副本，得到包含所有等级值的最终结果。这种方式不够简便，因为需要合并顺序不同的两个序列。
- **串行**：首先计算出一个变量的等级值，将原始数据包在其中，然后对包装后的数据中的另一个变量再求一次等级值。以这种方式生成的结构会比较复杂，但通过对最后结果做平铺处理可以缓解这个问题。

下面的函数创建了包含 y 值等级顺序的包装对象：

```
from typing import NamedTuple
class Ranked_Y(NamedTuple):
    r_y: float
    raw: Pair

def rank_y(pairs: Iterable[Pair]) -> Iterable[Ranked_Y]:
    return (
        Ranked_Y(rank, data)
        for rank, data in rank(pairs, lambda pair: pair.y)
    )
```

这里定义了一个包含 y 等级值以及原始数据的命名元组。rank_y()函数通过应用 rank()函数创建了这个元组的实例，而 rank()函数中使用了一个从 Pair 对象中提取 y 属性的匿名函数，最后得到了二元组实例。

因此当有如下输入时：

```
>>> data = (Pair(x=10.0, y=8.04),
...   Pair(x=8.0, y=6.95),
```

```
...    Pair(x=13.0, y=7.58),
etc.
...     Pair(x=5.0, y=5.68))
```

输出会如下所示：

```
>>> list(rank_y(data))
[Ranked_Y(r_y=1.0, raw=Pair(x=4.0, y=4.26)),
 Ranked_Y(r_y=2.0, raw=Pair(x=7.0, y=4.82)),
 Ranked_Y(r_y=3.0, raw=Pair(x=5.0, y=5.68)),
etc.
 Ranked_Y(r_y=11.0, raw=Pair(x=12.0, y=10.84))]
```

将原来的 Pair 对象包装进了包含等级值的新 Ranked_Y 对象中，但这不是我们想要的。在此基础上需要再次包装，得到包含 x 和 y 等级值的对象。

7.4.3 以多次包装代替状态变化

首先考虑创建一个名为 Ranked_X 的命名元组子类，包含 r_x 和 ranked_y 两个属性。其中 ranked_y 是一个 Ranked_Y 实例，Ranked_Y 包含两个属性：r_y 和 raw，虽然易于构建，但难以处理，因为 r_x 和 r_y 不在同一级结构中。下面介绍一个略复杂的包装过程，以得到简单的结果数据结构。

7

我们希望输出数据如下所示：

```
class Ranked_XY(NamedTuple):
    r_x: float
    r_y: float
    raw: Pair
```

该命名元组包含多个对等的属性，这种结构比深层嵌套结构易于处理。在某些应用中，需要多次变换数据，这里只有两次变换：对 x 和 y 求等级值。整个过程分为两步：首先像前面那样简单包装，然后是更通用的“拆包–再包装”。

基于 y 等级排序进行 x-y 等级排序如下所示：

```
def rank_xy(pairs: Sequence[Pair]) -> Iterator[Ranked_XY]:
    return (
        Ranked_XY(
            r_x=r_x, r_y=rank_y_raw[0], raw=rank_y_raw[1])
        for r_x, rank_y_raw in
            rank(rank_y(pairs), lambda r: r.raw.x)
    )
```

首先通过 rank_y()函数创建 Rank_Y 对象，然后对这些对象应用 rank()函数，基于原始数据中的 x 属性求等级值。该函数返回一个二元组：x 等级值和 Rank_Y 对象。最后基于 x 等级值 r_x、y 等级值 rank_y_raw[0]和原始对象 rank_y_raw[1]创建了 Ranked_XY 对象。

第二个函数展示了为元组添加数据的一种更通用的方法，Ranked_XY 对象的构建过程演示了如何从原有数据中拆包，并再次打包形成更复杂的结构，这是向元组中添加新变量的一种常用方法。

样本数据如下所示：

```
>>> data = (Pair(x=10.0, y=8.04), Pair(x=8.0, y=6.95),
... Pair(x=13.0, y=7.58), Pair(x=9.0, y=8.81),
etc.
... Pair(x=5.0, y=5.68))
```

可创建等级对象如下所示：

```
>>> list(rank_xy(data))
[Ranked_XY(r_x=1.0, r_y=1.0, raw=Pair(x=4.0, y=4.26)),
 Ranked_XY(r_x=2.0, r_y=3.0, raw=Pair(x=5.0, y=5.68)),
 Ranked_XY(r_x=3.0, r_y=5.0, raw=Pair(x=6.0, y=7.24)),
etc.
 Ranked_XY(r_x=11.0, r_y=10.0, raw=Pair(x=14.0, y=9.96))]
```

有了 x 和 y 的等级值，就可以计算斯皮尔曼等级顺序相关度了，最终可以根据原始数据算出斯皮尔曼等级相关系数。

前面的多次求等级值的方法涉及拆解元组并创建新的包含附加属性的单层元组，当需要从输入数据中计算多个目标值时，这是一种常用的方法。

7.4.4　计算斯皮尔曼等级顺序相关度

斯皮尔曼等级顺序相关度用于比较两个变量等级的相关性。它巧妙地避免了量级的影响，即使变量关系是非线性的，也能找到二者之间的关联，计算公式如下：

$$\rho = 1 - \frac{6\sum(r_x - r_y)^2}{n(n^2 - 1)}$$

首先计算每对观测值对应的等级值 r_x 和 r_y 差的平方和，对应的 Python 算法借助 sum() 函数和 len() 函数实现，如下所示：

```
def rank_corr(pairs: Sequence[Pair]) -> float:
    ranked = rank_xy(pairs)
    sum_d_2 = sum((r.r_x -r.r_y) ** 2 for r in ranked)
    n = len(pairs)
    return 1 - 6 * sum_d_2 / (n * (n ** 2 - 1))
```

前面用 Rank_XY 对象表示数据对，所以这里计算 r_x 和 r_y 属性的差值，然后取差值的平方和。

相关系数的具体含义请参阅统计学方面的资料。相关系数为 0 表示没有相关性，在散点图上

显示为无规律的随机数据点。值接近 1 或者−1 则表示的相关性很强，散点图上显示为一条清晰的直线或者曲线。

基于安斯库姆四重奏数据序列的示例如下：

```
>>> data = (Pair(x=10.0, y=8.04), Pair(x=8.0, y=6.95),
... Pair(x=13.0, y=7.58), Pair(x=9.0, y=8.81),
... Pair(x=11.0, y=8.33), Pair(x=14.0, y=9.96),
... Pair(x=6.0, y=7.24), Pair(x=4.0, y=4.26),
... Pair(x=12.0, y=10.84), Pair(x=7.0, y=4.82),
... Pair(x=5.0, y=5.68))
>>> round(pearson_corr( data ), 3)
0.816
```

可以看出该数据集的相关性很强。

第 4 章讲过如何计算皮尔逊相关系数，其中 `corr()`函数的输入值是两个无关的数值序列。下面用它来处理 `Pair` 对象序列：

```
import Chapter_4.ch04_ex4
def pearson_corr(pairs: Sequence[Pair]) -> float:
    X = tuple(p.x for p in pairs)
    Y = tuple(p.y for p in pairs)
    return ch04_ex4.corr(X, Y)
```

将 `Pair` 对象拆开后，作为输入传给 `corr()`函数，得到了另外一种相关系数：皮尔逊相关系数，以此比较两个变量的标准值。对于某些数据集来说，皮尔逊相关系数和斯皮尔曼相关系数差别不大，而对另外一些数据集来说，二者的差别会非常大。

在 EDA 中，同时使用多种统计工具是很重要的。要想知道具体原因，分别计算一下安斯库姆四重奏数据集的斯皮尔曼等级相关系数和皮尔逊相关系数就清楚了。

7.5 多态与类型匹配

一些函数式语言提供了多种方式来灵活处理静态函数类型定义，但问题是许多函数的数据类型是完全抽象的。以统计函数为例，只要保证做除法得到的商是 `numbers.Real` 的子类（例如 `Decimal`、`Fraction` 或者 `float` 类型），其函数定义对整数和实数是完全一致的。这些函数式语言提供了复杂的类型和类型匹配机制，使得编译器可以通过统一的抽象定义处理多种数据类型。Python 不存在这个问题，也不需要类型匹配。

Python 的实现方法与静态类型的函数式语言借助于（可能是）复杂的语言特征截然相反。基于参与运算的数据类型，Python 动态确定运算符的最终实现方式。使用 Python 编写的函数本就是抽象的，不与任何数据类型绑定，Python 运行时根据对象类型确定使用何种实现。关于从运算符到具体处理方法名称之间的映射细节，可参阅 Python 语言参考手册的 3.3.7 节以及类库中

numbers 模块的具体实现。

这意味着编译器并不能保证函数的输入数据和输出数据的类型是正确的，通常使用单元测试和 mypy 工具检测类型。

特殊情况下，当确实需要根据数据类型决定处理方法时，有如下两种解决方案：

- 使用 isinstance()函数确定具体属于哪种情况；
- 创建 numbers.Number 或者 NamedTuple 的子类并实现符合具体情况的多态方法。

有时需要两种方法兼用来纳入需要转换的所有数据类型，并通过 cast()函数为 mypy 显式指定数据类型。

前面等级排序的例子基于输入值是简单数据对的假设，虽然斯皮尔曼等级相关系数的函数定义满足该要求，但实践中经常需要计算多个变量间的所有相关系数。

下面抽象化等级排序，首先定义如下命名元组，包含等级值和原始数据项。

```
from typing import NamedTuple, Tuple, Any
class Rank_Data(NamedTuple):
    rank_seq: Tuple[float]
    raw: Any
```

这个类的典型应用场景如下所示：

```
>>> data = {'key1': 1, 'key2': 2}
>>> r = Rank_Data((2, 7), data)
>>> r.rank_seq[0]
2
>>>r.raw
{'key1': 1, 'key2': 2}
```

原始数据的每一行是一个字典，每个数据项包含两个等级值。应用既可以获取等级值，也可以获取原始数据本身。

然后为等级排序函数添加一些语法糖。前面许多例子使用了 for 语句处理可迭代对象和集合，但这里不会过多地使用 for 语句。在一些函数中，我们显式使用 iter()函数将集合转化为可迭代对象，这里使用 isinstance()函数进行类型检测，如下所示：

```
def some_function(seq_or_iter: Union[Sequence, Iterator]):
    if isinstance(seq_or_iter, Sequence):
        yield from some_function(iter(seq_or_iter), key)
        return
    # Do the real work of the function using the Iterator
```

通过类型检查序列和迭代器对象间的微小差别。这里使用 iter()函数将序列对象转化为迭代器，然后递归调用自身来处理数据。

这里使用了 Union[Sequence, Iterator]表示等级排序的数据结构，源数据排序后才能获得等级值，最简单的方法是用 list()函数将迭代器转化为序列。虽然仍使用 isinstance()函数，但不像前面那样基于序列生成迭代器，而是将迭代器转化为序列。

应尽量把等级排序函数定义得抽象一些。下面两个表达式定义了输入数据：

```
Source = Union[Rank_Data, Any]
Union[Sequence[Source], Iterator[Source]]
```

两类输入数据对应四种组合：

- ❑ Sequence[Rank_Data]
- ❑ Sequence[Any]
- ❑ Iterator[Rank_Data]
- ❑ Iterator[Any]

下面的 rank_data()函数分三种情况处理上面四种组合。

```
from typing import (
    Callable, Sequence, Iterator, Union, Iterable,
    TypeVar, cast, Union
)
K_ = TypeVar("K_") # Some comparable key type used for ranking.
Source = Union[Rank_Data, Any]
def rank_data(
        seq_or_iter: Union[Sequence[Source], Iterator[Source]],
        key: Callable[[Rank_Data], K_] = lambda obj: cast(K_, obj)
    ) -> Iterable[Rank_Data]:

    if isinstance(seq_or_iter, Iterator):
        # Iterator? Materialize a sequence object
        yield from rank_data(list(seq_or_iter), key)
        return

    data: Sequence[Rank_Data]
    if isinstance(seq_or_iter[0], Rank_Data):
        # Collection of Rank_Data is what we prefer.
        data = seq_or_iter
    else:
        # Convert to Rank_Data and process.
        empty_ranks: Tuple[float] = cast(Tuple[float], ())
        data = list(
            Rank_Data(empty_ranks, raw_data)
            for raw_data in cast(Sequence[Source], seq_or_iter)
        )

    for r, rd in rerank(data, key):
        new_ranks = cast(
            Tuple[float],
            rd.rank_seq + cast(Tuple[float], (r,)))
        yield Rank_Data(new_ranks, rd.raw)
```

前面把四类数据结构分为三种情况，下面详细说明。

- 如果输入是迭代器（没有实现__getitem__()方法的对象），把它实例化为一个列表对象，就都可以处理 Rank_Data 或其他类型的源数据了。这种情况包括 Iterable[Rank_Data] 和 Iterable[Any]类型对象。
- 如果输入是 Sequence[Any]类型对象，把这个未知对象包装进一个带有空等级值的 Rank_Data 元组中，创建一个 Sequence[Rank_Data]对象。
- 最后，如果输入是 Sequence[Rank_Data]类型对象，添加新的等级排序值到原有的 Rank_Data 容器中。

第一种情况递归调用 rank_data()自身，另外两种情况使用 rerank()函数基于算出的等级值构建新的 Rank_Data 对象。结构复杂的源数据将包含多个等级排序值。

为了消除等级元组数据类型的模糊性，需要使用复杂的 cast()表达式。可以用 mypy 工具提供的 reveal_type()函数调试编译器的类型推断。

rerank()函数与前面定义的 rank()函数略有不同，它返回包括等级值与源数据的二元组。

```
def rerank(
        rank_data_iter: Iterable[Rank_Data],
        key: Callable[[Rank_Data], K_]
    ) -> Iterator[Tuple[float, Rank_Data]]:
    sorted_iter = iter(
        sorted(
            rank_data_iter, key=lambda obj: key(obj.raw)
        )
    )
    # Apply ranker to head, *tail = sorted(rank_data_iter)
    head = next(sorted_iter)
    yield from ranker(sorted_iter, 0, [head], key)
```

rerank()的实现思路是对 Rank_Data 序列进行排序，其头部元素 head 用作 ranker()函数的种子，ranker()函数在剩余项中查找与头部元素匹配（等级计算标准相同）的其他对象，从而计算出一组匹配对象的等级值。

ranker()函数接收一组排序后的可迭代对象、一个基准值和一个等级值最小的排序对象集合，返回结果是一个由等级值和相应 Rank_Data 对象组成的二元组序列。

```
def ranker(
        sorted_iter: Iterator[Rank_Data],
        base: float,
        same_rank_seq: List[Rank_Data],
        key: Callable[[Rank_Data], K_]
    ) -> Iterator[Tuple[float, Rank_Data]]:
    try:
        value = next(sorted_iter)
    except StopIteration:
```

```
        dups = len(same_rank_seq)
        yield from yield_sequence(
            (base + 1 + base + dups) / 2, iter(same_rank_seq))
        return
    if key(value.raw) == key(same_rank_seq[0].raw):
        yield from ranker(
            sorted_iter, base, same_rank_seq + [value], key)
    else:
        dups = len(same_rank_seq)
        yield from yield_sequence(
            (base + 1 + base + dups) / 2, iter(same_rank_seq))
        yield from ranker(
            sorted_iter, base + dups, [value], key)
```

首先从排序后的 Rank_Data 组成的序列中取一个对象，如果出现 StopIteration 异常，说明数据源已空，没有待处理元素了。基于拥有相同等级值的 same_rank_seq 返回最终结果。

如果能取到下一个对象，就用 key()函数计算键值。如果与 same_rank_seq 中元素的键值相同，把它追加到当前相同等级值序列中。基于 sorted_iter 中的剩余数据、当前等级值、包含了 head 的新 same_rank_seq 以及 key()函数计算最终结果。

如果当前对象的键值与相同等级值集合中元素的 key 值不一致，则返回结果包含两部分。第一部分来自 same_rank_seq 中包含的相同等级值序列。第二部分来自排序后集合中的剩余元素，其基准值在当前相同等级值序列的基础上增加，初始化新的相同等级序列键值，key()函数保持不变。

ranker()函数使用 yield_sequence()输出结果，如下所示：

```
def yield_sequence(
        rank: float,
        same_rank_iter: Iterator[Rank_Data]
    ) -> Iterator[Tuple[float, Rank_Data]]:
    head = next(same_rank_iter)
    yield rank, head
    yield from yield_sequence(rank, same_rank_iter)
```

该实现强调了算法的递归含义，实践中应使用 for 语句进行优化。



运用尾调用优化技术将递归优化为循环时，首先要做好单元测试，确保递归版本通过单元测试后再开始优化。

接下来用前面定义的函数为数据标记（以及再次标记）等级值，从由简单标量组成的集合开始。

```
>>> scalars= [0.8, 1.2, 1.2, 2.3, 18]
>>> list(rank_data(scalars))
[Rank_Data(rank_seq=(1.0,), raw=0.8),
 Rank_Data(rank_seq=(2.5,), raw=1.2),
```

```
 Rank_Data(rank_seq=(2.5,), raw=1.2),
 Rank_Data(rank_seq=(4.0,), raw=2.3),
 Rank_Data(rank_seq=(5.0,), raw=18)]
```

源数据变成了 Rank_Data 对象的 raw 属性。

处理更复杂的数据时会出现多个等级值。下面是一个二元组序列：

```
>>> pairs = ((2, 0.8), (3, 1.2), (5, 1.2), (7, 2.3), (11, 18))
>>> rank_x = list(rank_data(pairs, key=lambda x:x[0]))
>>> rank_x
[Rank_Data(rank_seq=(1.0,), raw=(2, 0.8)),
Rank_Data(rank_seq=(2.0,), raw=(3, 1.2)),
Rank_Data(rank_seq=(3.0,), raw=(5, 1.2)),
Rank_Data(rank_seq=(4.0,), raw=(7, 2.3)),
Rank_Data(rank_seq=(5.0,), raw=(11, 18))]

>>> rank_xy = list(rank_data(rank_x, key=lambda x:x[1]))
>>> rank_xy
[Rank_Data(rank_seq=(1.0, 1.0), raw=(2, 0.8)),
Rank_Data(rank_seq=(2.0, 2.5), raw=(3, 1.2)),
Rank_Data(rank_seq=(3.0, 2.5), raw=(5, 1.2)),
Rank_Data(rank_seq=(4.0, 4.0), raw=(7, 2.3)),
Rank_Data(rank_seq=(5.0, 5.0), raw=(11, 18))]
```

首先定义了二元组集合，然后将二元组生成的 Rank_Data 对象序列标记等级值赋给 rank_x，接着对 Rank_Data 对象序列再次标记等级值，并赋给 rank_xy。

对 rank_corr()函数稍做修改，就可以基于 Rank_Data 对象的 rank_seq 属性计算任意序列的等级相关度了，具体修改留给读者练习。

7.6　小结

本章介绍了使用命名元组处理复杂数据结构的几种方法。命名元组的特点使它非常适用于函数式设计。它们由创建函数生成，可以通过位置标记或名称访问其属性。

本章还介绍了如何使用不可变的命名元组代替有状态的对象定义，其核心技术是将包裹在不可变的元组中的对象用作附加的属性值。

然后展示了 Python 中处理多种数据类型的方法，对于大部分数学运算，Python 能自动确定合适的实现方法，但在处理集合类型数据时，需要略微区别对待可迭代对象和序列。

接下来的两章将介绍 itertools 模块，这个库模块提供了许多处理可迭代对象的高级方法。其中许多工具是高阶函数，有助于实现简洁明了的函数式设计。

第8章

itertools 模块

8

函数式编程强调使用无状态对象，在 Python 中，可以使用生成器表达式、生成器函数和可迭代对象代替庞大的可变对象来达到这个目标。本章介绍如何使用 itertools 库中的函数来处理可迭代集合，这个库的许多函数有助于我们使用可迭代序列对象和集合对象。

第 3 章介绍过迭代器函数，本章将在此基础上进行拓展，另外还会用到第 5 章的一些相关函数。



有些函数从行为上看是普通的惰性 Python 可迭代对象，但从实现细节上看，它们在执行过程中创建了中间对象，消耗了大量内存。由于函数的实现细节会随着发布版本的变化而变化，这里不会提供针对某个具体函数的建议，但当程序出现性能或者内存问题时，记得仔细分析函数的具体实现方法。

“使用代码，朋友。”是我们一贯的建议[①]。

itertools 模块中包含大量迭代器函数，其中部分函数留待下一章讨论，本章要介绍的迭代器函数大体分为以下三大类。

- 处理无限迭代器的函数：适用于任何迭代器或者基于任何集合的迭代器。例如 enumerate() 函数不需要可迭代对象有上界。
- 处理有限迭代器的函数：这类函数或者对数据源做多次累积，或者生成数据源的归约。例如分组函数需要可迭代对象有上界。
- tee 迭代器函数：将一个迭代器克隆为多个副本，每个副本可以单独使用。这克服了 Python 迭代器只能使用一次的缺点。

虽然之前讲过，但这里再次强调可迭代对象的一个重要局限：只能使用一次。

① 原文“Use the source, Luke.”强调阅读源代码的重要性，化用了电影《星球大战》中的经典台词“Use the force, Luke.”。——译者注



可迭代对象只能使用一次。

这点听上去有点奇怪。当数值用尽后，迭代器中不再有数据，如果继续从其中取数据，将引发 `StopIteration` 异常。

除此之外，还有一些不太重要的限制，列举如下。

- 不能对可迭代对象使用 `len()` 函数。
- 可以用 `next()` 方法处理可迭代对象，这一点与容器不同。使用 `iter()` 函数将容器转化为迭代器，这样就可以使用 `next()` 方法了。
- `for` 语句自动对容器对象应用 `iter()` 函数,所以在 `for` 语句中容器和可迭代对象似乎没有区别。一个容器（例如列表）首先会转化为包含所有元素的迭代器；而一个可迭代对象，例如生成器，由于原本就符合迭代器协议，所以会返回自身。

以上几点是本章的主要背景知识。`itertools` 模块的核心思想，就是让用户不必过多关注可迭代对象的实现细节，且无须手动管理可迭代对象的复杂代码，而通过发挥可迭代对象的优势来编写出简洁明了的程序。

8.1　使用无限迭代器

`itertools` 模块提供的许多函数可以增强和扩展处理迭代数据源的能力，下面着重介绍以下三个函数。

- `count()`：`range()` 函数的无限版本。
- `cycle()`：循环迭代一组值。
- `repeat()`：按指定次数重复单个值。

本章旨在介绍在生成器表达式中使用这些函数，以及与生成器函数配合使用的方法。

8.1.1　用 count()计数

内置的 `range()` 函数需要定义范围的上界，而下限和步长值可选。`count()` 函数与之相反，需要给出起始值和一个可选的步长，无须定义上界。

可以用该函数定义类似于 `enumerate()` 这样的函数。例如可以通过 `zip()` 函数和 `count()` 函数定义 `enumertae()` 函数，如下所示：

```
enumerate = lambda x, start=0: zip(count(start), x)
```

将 `enumerate()` 函数定义为用 `zip()` 函数将 `count()` 函数与某个可迭代对象组合起来。

因此下面两个命令是等同的。

```
>>> list(zip(count(), iter('word')))
[(0, 'w'), (1, 'o'), (2, 'r'), (3, 'd')]
>>> list(enumerate(iter('word')))
[(0, 'w'), (1, 'o'), (2, 'r'), (3, 'd')]
```

二者都返回一系列二元组，元组的第一个元素是个整数计数器，第二个元素来自可迭代对象。在上面的例子中，迭代器构建于由字符组成的字符串。

zip()函数与 count()函数配合使用更简单，如下所示：

```
zip(count(1, 3), some_iterator)
```

count(b, s)返回序列$\{b, b+f, b+2f, b+3f, \cdots\}$。上面的例子首先创建了 1, 4, 7, 10, ⋯ 序列，然后用每个元素标记枚举器给出的值。enumerate()函数也有不足之处：不能改变迭代的步长。使用 enumerate()函数实现的版本如下：

```
((1 + 3 * e, x) for e, x in enumerate(some_iterator))
```

8.1.2 使用实数参数计数

count()函数接收非整形参数，可以定义 count(0.5, 0.1)这样的表达式来提供浮点数。如果步长值没有合适的表示形式，会形成累积误差。最好使用(0.5 + x * 0.1 for x in count())等整形 count()参数来避免累积误差出现。

下面介绍如何检验累积误差。探索实数近似值用到了函数式编程的一些实用技巧。

下面定义一个函数从迭代器中不断取值，直到满足给定条件。until()函数定义如下：

```
from typing import Callable, Iterator TypeVar
T_ = TypeVar("T_")
def until(
        terminate: Callable[[T_], bool],
        iterator: Iterator[T_]
    ) -> T_:
    i = next(iterator)
    if terminate(i):
        return i
    return until(terminate, iterator)
```

首先从迭代器对象中取一个值，类型由类型变量 T_定义。如果这个值通过了测试，即取到了符合要求的值，迭代结束，返回值的类型也是由类型变量 T_定义的；否则递归调用函数自身，测试后面的值。

下面是一个可迭代对象实例和一个测试函数。

```
Generator = Iterator[Tuple[float, float]]
source: Generator = zip(count(0, 0.1), (.1*c for c in count()))

Extractor = Callable[[Tuple[float, float]], float]
```

```
x: Extractor = lambda x_y: x_y[0]
y: Extractor = lambda x_y: x_y[1]

Comparator = Callable[[Tuple[float, float]], bool]
neq: Comparator = lambda xy: abs(x(xy)-y(xy)) > 1.0E-12
```

生成器 source 赋值语句中的类型标示表明它在二元组上迭代。两个抽取函数 x()和 y()从二元组中抽取实数。比较函数 neq()接收实数二元组，返回布尔值。

通过赋值语句创建匿名函数对象，其中的类型标示协助 mypy 工具确定参数和返回值类型。

mypy 工具检查 until()函数时，将类型变量 T_关联到实际类型 Tuple[float, float][①]。通过这个关联，能确认 source 生成器和 neq()函数可以作为 until()函数的参数。

until(neq, source)在每次迭代中比较实数的近似值，直到二者出现显著差异。count()函数给出其中一个近似值 $\sum_{x\in\mathbb{N}}.1$，生成器函数给出另一个近似值 $.1\times\sum_{x\in\mathbb{N}}1$。理论上这两个值没有差别，但基于抽象表示的具体近似值存在差异。

计算结果如下所示：

```
>>> until(neq, source)
(92.799999999999, 92.80000000000001)
```

928 次迭代后，累积误差达到了 10^{-12}，两个结果都没有精确的二进制表示。



count()函数的示例接近了 Python 的递归上限。如果要测试更大的累积误差，需要用尾调用优化技术重写 until()函数。

计算能检测到的最小误差的方法如下：

```
>>> until(lambda x, y: x != y, source)
(0.6, 0.6000000000000001)
```

用相等检测代替了误差范围检测。6 次迭代后，count(0, 0.1)的累积误差达到了 10^{-16}，用二进制表示 $\frac{1}{10}$ 需要无限长的位数，实际存储时只能截取保存有限位数字，导致误差产生并累积。

8.1.3　用 cycle()循环迭代

cycle()函数重复循环一组值，可用它循环数据集标识符对数据集进行分组。

还可以用它解决简单的 fizz-buzz 问题，关于该问题的多种解法可参考 http://rosettacode.org/wiki/FizzBuzz，基于它的一些有趣变体可参考 https://projecteuler.net/ problem=1。

① 这是从原始数据文件中记录数据行号的常规做法。——译者注

可以利用 cycle()函数生成一系列 True 和 False 值，如下所示：

```
m3 = (i == 0 for i in cycle(range(3)))
m5 = (i == 0 for i in cycle(range(5)))
```

返回结果是无限长的布尔值序列：[True, False, False, True, False, False, ...]或[True, False, False, False, False, True, False, False, False, False, ...]。

如果用 zip()处理有限个数值序列和上面两个标识序列，可以得到一个三元组序列，第一部分是数值，另外两个是分别代表是否为 3 的倍数或 5 的倍数的标识值。注意这里要引入一个有限的可迭代对象来生成目标数据的上限。这样的一系列值及其标识值如下：

```
multipliers = zip(range(10), m3, m5)
```

这样就得到了一个生成器对象，可以用 list(multipliers)查看其中包含的结果值，如下所示：

```
[(0, True, True), (1, False, False), (2, False, False), ..., (9, True, False)]
```

这样就可以拆解三元组了，用一个过滤器选出倍数值，舍弃其他数值：

```
total = sum(i
    for i, *multipliers in multipliers
    if any(multipliers)
)
```

for 从句将每个元组拆分成两部分：数值 i 和标识值 multipliers，如果标识值含真值，则保留该数值，否则将其舍弃。

该函数在 EDA 中很有用。

我们经常要处理大型数据集的样本，在最初的清洗和建模阶段，应使用小数据集，然后在更大的数据集上进行测试。使用 cycle()函数可以方便从大型数据集中选取记录。给定总体数据大小 N_p 与样本集大小 N_s，可算出循环体的大小。

$$c = \frac{N_p}{N_s}$$

假设用 csv 模块解析数据，则可以轻松生成子数据集，已知循环体大小 cycle_size 和两个打开的文件 source_file 和 target_file，生成子数据集的方法如下：

```
chooser = (x == 0 for x in cycle(range(cycle_size)))
rdr = csv.reader(source_file)
wtr = csv.writer(target_file)
wtr.writerows(
    row for pick, row in zip(chooser, rdr) if pick
)
```

首先基于拣选因子 cycle_size 和 cycle()函数生成拣选函数。比如总体数据集大小为

10 000 000，拣选数据集大小为 1000，即选取了 1/10 000 的数据。这里我们假设打开数据文件时，这些代码都嵌在 with 语句中。

接着用基于 cycle()函数的生成器表达式和取自 CSV 读取器的源数据文件过滤数据。由于用 chooser()函数和写数据函数都是非严格的，这类处理消耗内存很少。

稍后会介绍如何使用 compress()、filter()和 islice()函数实现相同的功能。

用这个方法还能将非标准 CSV 文件转换为标准 CSV 文件。只要数据解析器返回格式一致的元组，并通过写文件方法定义好 consumer 函数，就能用简单的脚本实现许多清洗和过滤操作。

8.1.4　用 repeat()重复单个值

repeat()函数的功能似乎有点奇怪：重复返回单个值。在需要单个值的时候可以用它替代 cycle()`函数。

选择所有数据和部分数据的差别在于：表达式(x==0 for x in cycle(range(size)))生成序列[True, False, False, ...]，用于选取部分数据；表达式(x==0 for x in repeat(0))生成序列[True, True, True, ...]，用于选取全部数据。

考虑如下代码：

```
all = repeat(0)
subset = cycle(range(100))
choose = lambda rule: (x == 0 for x in rule)
# choose(all) or choose(subset) can be used
```

只需简单修改参数，就可以切换选择全部数据或部分数据。扩展该方法可实现随机选取数据集，如下所示：

```
def randseq(limit):
    while True:
        yield random.randrange(limit)
randomized = randseq(100)
```

randseq()函数在指定范围内生成无限长的随机数序列，丰富了的 cycle()和 repeat()两种拣选模式。

不同拣选方法的实现如下：

```
[v for v, pick in zip(data, choose(all)) if pick]
[v for v, pick in zip(data, choose(subset)) if pick]
[v for v, pick in zip(data, choose(randomized)) if pick]
```

使用 chose(all)，chose(subset)或者 chose(randomized)可以方便地为后续分析提供输入数据。

8.2 使用有限迭代器

itertools 模块包含许多用于生成有限序列的函数，下面介绍其中 10 个函数，以及几个相关的内置函数。

- enumerate()：实际上该函数在__builtins__包中，但可以应用于迭代器，且用法与 itertools 模块中的其他函数类似。
- accumulate()：返回作为输入的可迭代对象的归约序列，它是高阶函数，能完成很多复杂的计算。
- chain()：将多个可迭代对象按顺序组合在一起。
- groupby()：将输入的可迭代对象按照指定函数分割为多个子迭代对象。
- zip_longest()：将多个可迭代对象中的元素合并在一起，内置的 zip()函数按照最短输入值进行截断，zip_longest()函数则对较短的可迭代对象插入填充值。
- compress()：根据第二个布尔可迭代对象形式的输入参数对第一个可迭代对象进行筛选。
- islice()：序列切片函数的可迭代对象版本。
- dropwhile()和 takewhile()：使用布尔函数过滤可迭代对象。与 filter()和 filterfalse()这类过滤函数不同，布尔值会影响对后续所有值的筛选。
- filterfalse()：对可迭代对象应用过滤函数，其返回结果与 filter()函数的返回结果相反。
- starmap()：将函数应用于由元组组成的可迭代序列。序列的每个元素以*args 的形式作为函数的参数，map()函数通过提供多个并列的可迭代参数可以实现相同的功能。

之前根据对可迭代对象进行的重组、过滤和映射操作，大致将这些函数分成了三类。

8.2.1 用 enumerate()添加序号

第 7 章使用 enumerate()函数将等级值赋给排序后的数据。可以将一个值与它在原始序列中的位置序号组对，如下所示：

```
pairs = tuple(enumerate(sorted(raw_values)))
```

将 raw_values 中的数据排序，生成一个升序排列的二元组序列，然后将它实例化，以备后续计算使用。表达式执行结果如下所示：

```
>>> raw_values = [1.2, .8, 1.2, 2.3, 11, 18]
>>> tuple(enumerate( sorted(raw_values)))
((0, 0.8), (1, 1.2), (2, 1.2), (3, 2.3), (4, 11), (5, 18))
```

第 7 章实现的另一个版本的枚举函数 rank()处理数据的方式更符合统计计算的要求。

上面这种方法是解析原始数据文件来记录数据行号的常见做法。很多时候我们需要创建某种

形式的 row_iter()函数从原始数据文件中提取字符串，例如遍历 XML 文件的标签或 CSV 文件列中的字符串，有时甚至需要通过 Beautiful Soap 库从 HTML 文件中解析数据。

第 4 章通过解析 XML 文件生成简单的位置元组序列，然后生成了包含起点、终点和距离的路径段，但没有给路径段编号，因此当把路径段排序后，就无法获得路径段的原始顺序信息了。

第 7 章拓展了基本解析器，创建了代表旅行路途中每个路径段的命名元组，这个增强版解析器的输出如下所示：

```
(Leg(start=Point(latitude=37.54901619777347, longitude=
 -76.33029518659048), end=Point(latitude=37.840832, longitude=
 -76.273834), distance=17.7246),
 Leg(start=Point(latitude=37.840832, longitude=-76.273834),
 end=Point(latitude=38.331501, longitude=-76.459503),
 distance=30.7382),
 Leg(start=Point(latitude=38.331501, longitude=-76.459503),
 end=Point(latitude=38.845501, longitude=-76.537331),
 distance=31.0756),
...,
 Leg(start=Point(latitude=38.330166, longitude=-76.458504),
 end=Point(latitude=38.976334, longitude=-76.473503),
 distance=38.8019))
```

其中第一个 Leg 对象表示 Chesapeake Bay 两地间的短途旅行。

下面创建一个能构造出更复杂元组的函数，将序号信息也加入元组中。首先定义一个更复杂的 Leg 类：

```
from typing import NamedTuple
class Point(NamedTuple):
    latitude: float
    longitude: float

class Leg(NamedTuple):
    start: Point
    end: Point
    distance: float
```

基本与第 7 章中的 Leg 定义相同，在保持原有属性的基础上增加了序号属性。下面定义函数拆解数据对并生成 Leg 实例。

```
from typing import Iterator

def ordered_leg_iter(
        pair_iter: Iterator[Tuple[Point, Point]]
    ) -> Iterator[Leg]:
    for order, pair in enumerate(pair_iter):
        start, end = pair
        yield Leg(
            order,
```

```
            start,
            end,
            round(haversine(start, end), 4)
        )
```

使用该函数枚举包含起点和终点的各对数据。拆解每个数据对，重新组合序号、起点、终点以及 `haversine(start, end)`参数，形成新的 `Leg` 实例，该生成器函数的输入值是由数据对组成的可迭代序列。

根据前面的说明，该函数的用法如下所示：

```
filename = "file:./Winter%202012-2013.kml"
with urllib.request.urlopen(filename) as source:
    path_iter = float_lat_lon(row_iter_kml(source))
    pair_iter = legs(path_iter)
    trip_iter = ordered_leg_iter( pair_iter )
    trip = list(trip_iter)
```

首先将原始数据解析为路径点，然后基于 `Leg` 对象生成旅行数据。这里用 `enumerate()`函数确保为每个元素分配一个从 0 开始依次递增且不会重复的序号。可以通过设置第二个参数来设置序号的起始值。

8.2.2　用 accumulate()计算汇总值

`accumulate()`函数基于给定的函数返回一个可迭代对象，将一系列归约值汇总在一起，遍历迭代器得出当前汇总值。默认的函数是 `operator.add()`。可以使用乘积函数代替默认的求和函数来改变 `accumulate()`的行为。Python 文档中有一个很巧妙的实例，使用 `max()`函数得到当前序列中的最大值。

可使用当前汇总值计算数据四等分值，计算每个样本值的当前汇总值，并用公式 `int(4 * value / total)`把它们四等分。

8.2.1 节讲过用一系列经纬度坐标描述旅途的一系列路径段。基于距离把路径点四等分，可以找到旅行路线的中点。

旅行数据格式如下：

```
(Leg(start=Point(latitude=37.54901619777347, longitude=-76.33029518659048),
     end=Point(latitude=37.840832, longitude=-76.273834), distance=17.7246),
 Leg(start=Point(latitude=37.840832, longitude=-76.273834),
     end=Point(latitude=38.331501, longitude=-76.459503), distance=30.7382),
...,
 Leg(start=Point(latitude=38.330166, longitude=-76.458504),
     end=Point(latitude=38.976334, longitude=-76.473503), distance=38.8019))
```

每个路径段对象包含起点、终点和距离 3 个元素，四等分算法如下所示：

```
distances = (leg.distance for leg in trip)
distance_accum = tuple(accumulate(distances))
total = distance_accum[-1] + 1.0
quartiles = tuple(int(4 * d / total) for d in distance_accum)
```

从路径段中提取距离数据，计算每段的累积距离，该序列的最后一项便是总距离，在总距离上加 1.0 确保 `4 * d / total` 返回 3.9983，然后截断取到 3。如果不加 1.0，最后一项的值会为 4，从而产生一个不存在的“第五等分”。对某些数据（数值非常大）来说，可能需要加上一个更大的数值。

`quartiles` 的计算结果如下所示：

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

可以使用 `zip()`函数合并等分序列值与原始数据点，还可以使用类似于 `groupby()`的函数为每个等分创建路径段集合。

8.2.3　用 **chain()**组合多个迭代器

可以用 `chain()`函数将多个迭代器组合为单个迭代器，比如将被 `groupby()`函数分开的数据重新组合起来。对于简单集合，用这个方法可以一次处理多个集合。

如果需要在一个简单迭代序列中处理多个文件中的数据，可以使用 `chain()`函数结合 `contextlib.ExitStack()`方法来实现，具体做法如下：

```
from contextlib import ExitStack
import csv
def row_iter_csv_tab(*filenames: str) -> Iterator[List[str]]:
    with ExitStack() as stack:
        files = [
            stack.enter_context(cast(TextIO, open(name, 'r')))
            for name in filenames
        ]  # type: List[TextIO]
        readers = map(
            lambda f: csv.reader(f, delimiter='\t'),
            files)
        yield from chain(*readers)
```

首先创建了一个包含多个打开的上下文的 `ExitStack` 对象。当 `with` 语句执行完毕后，`ExitStack` 对象中所有打开的对象都会合理地关闭，所以在 `ExitStack` 对象中创建了多个打开的文件对象。

基于 `files` 变量中的多个文件对象，我们创建了一系列 CSV readers 对象，保存在 `readers` 变量中。所有文件都是以 Tab 分隔的，因此便于用一个简单的函数处理所有文件。

也可以用如下方式打开一系列文件：

```
readers = [csv.reader(f, delimiter='\t') for f in files]
```

最后，通过 chain(*readers)将多读取器合并成单个迭代器，包含所有文件的行序列数据。

注意，这里不能用 return 返回 chain(*readers)，如果使用 return，函数将退出 with 语句，关闭所有文件，所以这里必须用 yield 返回某个文件的所有行，同时不退出 with 语句。

8.2.4 用 groupby()切分迭代器

通过对每个元素应用 key 函数进行求值，groupby()函数将一个迭代器切分为多个小迭代器。如果后一个元素的 key 值等于前一个元素的 key 值，会将这两个元素放在同一个分组中；如果与前一个元素的 key 值不同，则当前分组结束，将当前元素放到新的分组中。

groupby()函数的输出是一系列二元组，每个元组包括一个 key 值和包含该组元素的迭代器，该迭代器的元素可以通过转换保存为元组，也可以归约为汇总值。这种情况下将不会保留迭代器中的值。

8.2.2 节介绍了如何计算输入序列的四等分值。

基于旅行原始数据和算出的四等分值，可如下所示对数据进行分组：

```
group_iter = groupby(
    zip(quartile, trip),
    key=lambda q_raw: q_raw[0])
for group_key, group_iter in group_iter:
    print(group_key, tuple(group_iter))
```

首先用 zip()函数将四等分值和原始数据组合在一起，返回包含二元组的迭代器，然后用 groupby()函数基于给定的匿名函数按照四等分值对数据进行分组，最后用 for 循环检查 groupby()函数的返回结果。整个过程展示了获得键值组以及包含组成员的迭代器的方法。

groupby()函数的输入中的 key 值必须是排序好的，以确保分在一组中的元素是相邻的。

请注意，也可以使用 defaultdict(list)方法创建分组，实现方法如下：

```
from collections import defaultdict
from typing import Iterable, Callable, Tuple, List, Dict

D_ = TypeVar("D_")
K_ = TypeVar("K_")
def groupby_2(
        iterable: Iterable[D_],
        key: Callable[[D_], K_]
    ) -> Iterator[Tuple[K_, Iterator[D_]]]:
    groups: Dict[K_, List[D_]] = defaultdict(list)
```

```
    for item in iterable:
        groups[key(item)].append(item)
    for g in groups:
        yield g, iter(groups[g])
```

首先创建了一个 defaultdict 类,并为每个键值关联一个 list 对象。类型标示指出,key()函数将任意数据类型转化为键值类型 K_，与字典中的键值类型 K_是一致的。

每个数据项通过 key()函数生成键值，数据项本身被追加到该键值对应的 defaultdict 类列表中。

完成对所有数据项的分组后，就可以按照每个共同的 key 以迭代方式返回每个分组了，这与 groupby()函数的行为类似。由于作为输入的迭代器的排序结果不一定一致，分组结果可能包含相同的元素，但顺序可能不同。

类型标示中，源数据的类型是 D_，返回结果是一个包含 D_型迭代器的迭代器，表明函数没有执行映射操作，输入数据类型与范围对象的类型完全一致。

8.2.5 用 zip_longest()和 zip()合并迭代器

第 4 章用到了 zip()函数。zip_longest()函数与它的区别在于，zip()返回结果的长度是输入参数中最短序列的长度，而 zip_longest()为较短的序列填充值，直到遍历完最长的序列。

可以通过指定 zip_longest()函数的 fillvalue 参数代替默认的填充值 None。

EDA 的大多数应用场景都不需要填充默认值，虽然 Python 标准库文档提供了 zip_longest()函数的几个用例，但除此之外，本书关注的数据分析领域内难寻其适合的使用场景。

8.2.6 用 compress()过滤

内置的 filter()函数使用谓词来确定对某个数据项的取舍。除了使用函数进行计算，也可以使用第二个可迭代对象确定对元素的取舍。

filter()函数定义如下：

```
def filter(function, iterable):
    i1, i2 = tee(iterable, 2)
    return compress(i1, map(function, i2))
```

首先用 tee()函数克隆出可迭代对象的两个副本（稍后会详细介绍该函数），map()函数将谓词函数 function()映射至可迭代对象的每个值，形成由 True 和 False 组成的布尔值序列，保留其中与 True 关联的值,实现对源数据的过滤。这样就基于 compress()函数实现了 filter()函数的效果。

8.1.3 节使用了一个简单的生成器表达式拣选数据，其关键部分如下：

```
choose = lambda rule: (x == 0 for x in rule)
keep = [v for v, pick in zip(data, choose(all)) if pick]
```

规则的每个值都是一个能生成布尔值序列的函数。选择所有值的规则对应全为 `True` 的序列，选择固定子数据集则需要在 `True` 值和 $c-1$ 个 `False` 值间循环。

如果用 `compress(some_source, choose(rule))`代替上面的列表解析，处理过程可简化为：

```
compress(data, choose(all))
compress(data, choose(subset))
compress(data, choose(randomized))
```

这些示例使用了前面定义的拣选规则：`all`[①]、`subset` 和 `randomized`，其中 `subset` 和 `randomized` 需要定义合适的系数从源数据中取出 $\frac{1}{c}$ 行。`choose` 表达式基于其中一条拣选规则定义一个布尔值可迭代对象。通过将该行选择可迭代序列应用于源数据来选择行。

以上实现都是非严格的，只在需要时才从数据源中读取行数据，从而高效处理海量数据。另外，简洁的 Python 代码让我们无须编写复杂的配置文件和解析器，便可以灵活选择拣选规则，用一小段代码就可以完成大型数据取样应用的配置。

8.2.7 用 `islice()`选取子集

第 4 章使用了切片符号从集合中选取子集，当时是将一个列表对象中的元素组对。例如对于下面这个简单的列表：

```
flat = ['2', '3', '5', '7', '11', '13', '17', '19',
 '23', '29', '31', '37', '41', '43', '47', '53',
 '59', '61', '67', '71',
...
 ]
```

可以通过列表切片创建数据对：

```
>>> list(zip(flat[0::2], flat[1::2]))
[(2, 3), (5, 7), (11, 13), ...]
```

借助 `islice()`函数则无须实例化列表对象就能实现相同的功能，处理任意大小的可迭代对象，如下所示：

```
flat_iter_1 = iter(flat)
flat_iter_2 = iter(flat)
```

① 在 8.1.4 节定义的，不是内置函数 `all`。——译者注

```
zip(
    islice(flat_iter_1, 0, None, 2),
    islice(flat_iter_2, 1, None, 2)
)
```

这里基于一维数据点列表创建了两个独立的迭代器，它们可能是遍历打开的文件或者数据库结果集的两个迭代器。两个迭代器必须彼此独立，以保证一个 `islice()`函数的变动不会影响另一个 `islice()`函数。

`islice()`函数的两组输入参数与 `flat[0::2]`和 `flat[1::2]`表达式类似。由于没有切片符号式的简写形式，必须指明起始参数和结束参数，步长值可以省略，而取默认值 1。上面的代码返回由二元组组成的数据序列：

```
[(2, 3), (5, 7), (11, 13), (17, 19), (23, 29),
...
(7883, 7901), (7907, 7919)]
```

由于 `islice()`函数返回迭代器，所以可用于处理巨大的数据集，例如从一个大型数据集中提取一个子集。除了使用 `filter()`函数和 `compress()`函数，还可以用 `islice(source, 0, None, c)`从大型数据集中提取 $\frac{1}{c}$ 个数据项。

8.2.8　用 dropwhile()和 takewhile()过滤状态

`dropwhile()`和 `takewhile()`是有状态的过滤函数：从一种模式开始，当满足给定的谓词函数时切换到另一模式。`dropwhile()`函数开始采用拒绝模式，当谓词函数变为 `False` 时切换为通过模式。`takewhile()`则从通过模式开始，当谓词函数变为 `False` 时切换到拒绝模式。二者都是过滤器，所以会处理整个可迭代对象。

可以利用这些函数过滤掉输入文件的头部和尾部。首先用 `dropwhile()`过滤掉文件头部的文本行，返回剩余的数据，然后用 `takewhile()`保留数据部分，去掉文件尾部的文本行。回到第 3 章中的 GPL 文件格式，文件头部如下所示：

```
GIMP Palette
Name: Crayola
Columns: 16
#
```

其后的文本行如下所示：

```
255    73 108      Radical Red
```

基于 `dropwhile()`函数，可以方便地定位到文件头部的最后一行，即#这一行，如下所示：

```
with open("crayola.gpl") as source:
    rdr = csv.reader(source, delimiter='\t')
    rows = dropwhile(lambda row: row[0] != '#', rdr)
```

之前创建了 CSV reader 基于 Tab 字符解析文本行，可以方便地将名称和颜色三元组分开，之后进一步解析三元组，就可得到#行之后处理文件剩余内容的迭代器了。

可以使用 islice()函数去掉可迭代对象的第一项，然后解析颜色的细节，如下所示：

```
color_rows = islice(rows, 1, None)
colors = (
    (color.split(), name) for color, name in color_rows
)
print(list(colors))
```

表达式 islice(rows, 1, None)与 rows[1:]这样的切片表达式类似，会丢弃第一项。一旦移除头部的文本行，就可以解析颜色元组并返回更有用的颜色对象了。

就该文件而言，还可以通过 CSV reader 解析出的列数实现文本过滤，例如使用 dropwhile(lambda row: len(row) == 1, rdr)方法也可以过滤掉文件头部，但这种方法并不普适，通常通过定位头部最后一行比通过一些文本特征区别文件头和数据更容易。本例中文件头部是通过列数识别出来的，属于个例。

8.2.9　基于 filterfalse()和 filter()的两种过滤方法

8

第 5 章用到了 filter()函数，可以基于它定义 itertools 模块中的 filterfalse()函数，如下所示：

```
filterfalse = (lambda pred, iterable:
    filter(lambda x: not pred(x), iterable)
)
```

基于 filter()函数定义意味着谓词函数可以是 None。函数 filter(None, iterable)返回可迭代对象中的所有真值，函数 filterfalse(None, iterable)则返回所有非真值。

```
>>> filter(None, [0, False, 1, 2])
>>> list(_)
[1, 2]

>>> filterfalse(None, [0, False, 1, 2])
<itertools.filterfalse object at 0x101b43a50>
>>> list(_)
[0, False]
```

使用 filterfalse()函数意在提高代码复用性。使用这个简洁的函数，可以方便地将输入数据分为“通过”和“舍弃”两组，而无须使用逻辑非这种繁复的表达方式。

该思想体现如下：

```
iter_1, iter_2 = iter(some_source), iter(some_source)
good = filter(test, iter_1)
bad = filterfalse(test, iter_2)
```

显然，这会包含输入数据中所有的值。test()函数无须变动，从而避免了误用括号造成难以察觉的逻辑错误。

8.2.10 将 starmap()和 map()应用于数据

Python 内置的 map()函数是高阶函数，能对可迭代对象中的每个元素进行映射。简易版的 map()函数如下所示：

```
map = (lambda function, arg_iter:
    (function(a) for a in arg_iter)
)
```

若 arg_iter 为可迭代对象，以上定义完全正确，但实际的 map()函数比这里的实现更复杂，能处理多个可迭代对象。

itertools 模块的 starmap()函数实际上是 map()函数的*a 版本，如下定义：

```
starmap = (lambda function, arg_iter:
    (function(*a) for a in arg_iter)
)
```

相比于 map()函数，该实现方法在语义上有微小的变化，使之更适于处理嵌套元组式的数据结构。

map()函数也可以接收多个可迭代对象作为输入参数，这些可迭代对象被 zip 在一起，行为类似于 startmap()。由源可迭代对象组成的各个元素构成了给定函数的输入参数。

可以通过下面两种表达式定义 map(function, iter1, iter2, ..., itern)：

```
map1 = (lambda function, *iters:
    (function(*args) for args in zip(*iters))
)

map2 = (lambda function, *iters:
    (starmap(function, zip(*iters)))
)
```

不同迭代器中的值通过 zip(*iters)组成参数元组，然后通过*args 结构体展开为给定函数的参数列表，所以可以基于更抽象的 starmap()函数构建出 map()函数。

理解了上面的计算过程，就可以基于 startmap()函数重新定义旅行数据的计算过程了。在创建 Leg 对象之前，首先创建位置点对，每个数据对如下所示：

```
((Point(latitude=37.54901619777347, longitude=-76.33029518659048),
  Point(latitude=37.840832, longitude=-76.273834)),
...,
 (Point(latitude=38.330166, longitude=-76.458504),
```

```
    Point(latitude=38.976334, longitude=-76.473503))
)
```

接着通过 starmap()函数组合生成 Leg 对象，如下所示：

```
from Chapter_7.ch07_ex1 import float_lat_lon, Leg, Point
from Chapter_6.ch06_ex3 import row_iter_kml
from Chapter_4.ch04_ex1 import legs, haversine
from typing import List, Callable

make_leg = (lambda start, end:
    Leg(start, end, haversine(start,end))
)  # type: Callable[[Point, Point], Leg]
with urllib.request.urlopen(url) as source:
    path_iter = float_lat_lon(row_iter_kml(source))
    pair_iter = legs(path_iter)
    trip = list(starmap(make_leg, pair_iter))
```

make_leg()函数的输入是一对 Point 对象，返回值是包含起点、终点和距离的 Leg 对象。第 4 章中定义的 legs()函数生成了包含一段路径起点和终点的 Point 对象二元组，用作 make_leg()的输入，并最终生成 Leg 对象。

使用 starmap(function, some_list)方法避免了写出(function(*args) for args in some_list)这样冗长的生成器表达式，也更易读。

8.3 使用 tee()函数克隆迭代器

使用 tee()函数可以突破 Python 处理可迭代对象时的一条重要规则，这条规则非常重要，在此重申一遍。



迭代器只能使用一次。

可以使用 tee()函数克隆可迭代对象。我们可以利用它多次使用序列中的数据而不必实例化序列，例如对一个大型数据集求算术平均值的写法如下所示：

```
def mean(iterator: Iterator[float]) -> float:
    it0, it1 = tee(iterator, 2)
    N = sum(1 for x in it0)
    s1 = sum(x for x in it1)
    return s1 / N
```

计算过程不涉及任何实例化数据集的操作，从而避免了将整个数据集放入内存中。注意，类型标示 float 并不排除使用整数的场景，mypy 能处理类型转换规则，这样的定义兼顾了整数和实数类型参数。

看上去不错，但实际使用 tee()函数存在一个严重的限制条件。在绝大多数 Python 实现中，克隆是通过实例化序列实现的。处理小数据集时，它确实有助于我们突破“只能使用一次”的限制，但它对于大型数据集的效果往往不佳。

另外，目前 tee()函数的实现方法需要遍历源迭代器。对使用者来说，如果有无限次使用迭代器的语法糖会很方便，但在实际使用中，这样的实现其实很难驾驭。所以在 Python 中使用 tee()函数时要小心。

8.4　itertools 模块代码范例

Python 库文档的“itertools”章的“Itertools Recipes”部分包含了使用该模块函数的很多范例。这里不重复文档中的内容，而是直接引用。学习 Python 函数式编程，这部分文档属于必读内容。



Python 标准库文档“itertool”章的“Itertools Recipes”节是很好的学习资源，见 https://docs.python.org/3/library/itertools.html#itertools-recipes。

注意，这里列出的范例都不是 itertools 模块中可以导入的函数，如果想在自己开发的应用中使用这些范例，需要阅读代码并理解原理，然后通过复制、修改来使用代码。

下表总结了基于 itertools 模块中的基本函数实现的一些自定义函数范例。

函　数　名	参数列表	返回结果
take	(n, iterable)	以列表形式返回作为输入参数的可迭代对象的前 *n* 个值。基于 islice()函数实现
tabulate	(function, start=0)	返回 function(0), function(1), …。通过 map(function, count())实现
consume	(iterator, n)	使可迭代对象前进 *n* 步，如果 *n* 值为 None，则遍历完可迭代对象，返回空列表
nth	(iterable, n, default=None)	返回可迭代对象的第 *n* 个值，如果取不到该值则返回给定的默认值。基于 islice()函数实现
quantify	(iterable, pred=bool)	对可迭代对象的每个元素应用谓词函数，返回结果为“真”的元素个数。基于 sum()函数和 map()函数实现，并利用了布尔值 True 转换为整数时值为 1 的特点
padnone	(iterable)	返回输入可迭代对象中的值，并在后面追加无限个 None 值。主要用于创建类似于 zip_longest()或者 map()的函数
ncycles	(iterable, n)	返回 *n* 次可迭代对象中的值
dotproduct	(vec1, vec2)	数学上向量点积的定义：两个向量对应分量乘积之和
flatten	(listOfLists)	展开一层嵌套列表，将多个列表链接为一个列表
repeatfunc	(func, times=None, *args)	根据输入的参数列表 args 执行 times 次函数 func
pairwise	(iterable)	s -> (s0, s1), (s1, s2), (s2, s3)

（续）

函 数 名	参数列表	返回结果
grouper	(iterable, n, fillvalue=None)	将输入数据按固定长度分组
roundrobin	(*iterables)	roundrobin('ABC', 'D', 'EF') --> A D E B F C
partition	(pred, iterable)	根据谓词函数将输入可迭代对象划分为真值部分和非真值部分
unique_everseen	(iterable, key=None)	按照出现顺序列出所有元素，之前出现过的元素除外
unique_justseen	(iterable, key=None)	按照出现顺序，列出与前一个元素不同的元素。多用于为已排序的列表剔除相同元素
iter_except	(func, exception, first=None)	重复执行函数 func，直到出现异常。常用于遍历操作，直到发生 KeyError 或者 IndexError

8.5 小结

本章介绍了 itertools 模块中的部分函数，这些函数有助于我们更好地使用可迭代对象。

首先介绍了无限迭代器，重复执行而不停止，包括 count()、cycle()和 repeat()函数。由于它们自身不会终止，使用它们的函数必须确定何时停止接收数值。

然后介绍了有限迭代器，包括内置函数和 itertools 模块中的函数。这些函数接收可迭代对象作为输入参数，所以当该对象中不再有值时，函数执行完毕，包括 enumerate()、accumulate()、chain()、groupby()、zip_longest()、zip()、compress()、islice()、dropwhile()、takewhile()、filterfalse()、filter()、starmap()和 map()。使用这些函数，让我们可以用简洁的函数代替复杂的生成器表达式。

最后介绍了官方文档中的部分范例及其实现方法，它们都是常用的设计模式。你可以尝试在自己开发的应用程序中使用这些范例。

第 9 章将介绍 itertools 模块中与排列组合相关的函数。与本章出现的函数相比，这些函数不适于处理大型数据集，属于不同的迭代处理工具。

第 9 章

高级 itertools 技术

函数式编程强调程序的无状态性，在 Python 中，这意味着开发者应使用生成器表达式。本章继续介绍 itertools 库中用于处理可迭代集合的函数。

上一章中的迭代器函数大致分为以下三类。

- 无限迭代器函数：适用于任何可迭代对象，或者基于任何集合的迭代器，处理数据源中的所有元素。
- 有限迭代器函数：多次汇聚数据源，或者生成数据源的归约。
- tee()函数：将一个迭代器克隆多份，每个副本可独立使用。

本章介绍 itertools 模块中与排列组合有关的函数，以及基于它们的范例函数，这些函数包括：

- product()　返回输入集合的笛卡儿积，相当于嵌套 for 循环；
- permutations()　返回 p 维空间中所有长度为 r 的包含所有可能顺序的元组，不含重复元素；
- combinations()　返回 p 维空间中所有长度为 r 的有序元组，不含重复元素；
- combinations_with_replacement()　返回 p 维空间中所有长度为 r 的有序元组，包含重复元素。

使用这些函数可以构建输入为小数据集、输出为大型数据集的算法。许多问题的解只能在规模巨大的排列空间中通过穷举的方式找到，这些函数有助于我们以比较简单的方式生成排列空间，尽管在某些场景中，简单解并非最优解。

9.1　笛卡儿积

"笛卡儿积"指基于一组集合生成所有可能的元素组合。

从数学角度看，两个集合的积$\{1, 2, 3, \cdots, 13\} \times \{C, D, H, S\}$是一个长度为 52 的二元组集合。

```
{(1, C), (1, D), (1, H), (1, S),
 (2, C), (2, D), (2, H), (2, S),
...,
 (13, C), (13, D), (13, H), (13, S)}
```

运行如下命令可以得到解：

```
>>> list(product(range(1, 14), '♣♦♥♠'))
[(1, '♣'), (1, '♦'), (1, '♥'), (1, '♠'),
 (2, '♣'), (2, '♦'), (2, '♥'), (2, '♠'),
...
 (13, '♣'), (13, '♦'), (13, '♥'), (13, '♠')]
```

可以对任意多个可迭代集合开展乘积运算。参与计算的集合越多，结果集越大。

9.2　对积进行归约

在关系数据库理论中，可以把表间的 `join` 操作看作带过滤条件的笛卡儿积。一条 SQL 语句中，如果 `SELECT` 语句后面没有 `WHERE` 从句，返回结果就是表中记录的笛卡儿积，或者说不带过滤条件的乘积运算是糟糕的算法。枚举所有的组合，再通过过滤保留符合条件的组合，可以通过 `itertools` 模块的 `product()`函数实现。

可以通过下面的函数定义两个可迭代集合或者生成器间的 `join` 操作。

```
JT_ = TypeVar("JT_")
def join(
        t1: Iterable[JT_],
        t2: Iterable[JT_],
        where: Callable[[Tuple[JT_, JT_]], bool]
    ) -> Iterable[Tuple[JT_, JT_]]:
    return filter(where, product(t1, t2))
```

可迭代对象 `t1` 和 `t2` 的所有组合都参与计算。`filter()`函数通过给定的 `where()`函数确定对类型为 `Tuple[JT_, JT_]`的二元组的取舍。其中 `where()`函数类型是 `Callable[[Tuple[JT_, JT_]], bool]`，说明返回值是布尔值。当数据库中没有任何可用的索引或者顺序标记时，SQL 查询只能在这种不理想的场景中低效运作。

这种算法实现虽然可用，但效率很低，通常需要仔细研究问题和数据，寻找更高效的算法。

首先稍微抽象一下问题，用多个数据项间最大/最小距离查找问题代替简单的布尔值匹配，比较的结果是一个实数。

假设有如下由 `Color` 对象组成的数据集：

```
from typing import NamedTuple
class Color(NamedTuple):
    rgb: Tuple[int, int, int]
```

```
    name: str
[Color(rgb=(239, 222, 205), name='Almond'),
 Color(rgb=(255, 255, 153), name='Canary'),
 Color(rgb=(28, 172, 120), name='Green'),...
 Color(rgb=(255, 174, 66), name='Yellow Orange')]
```

更多信息可参考第6章中解析颜色文件生成 `namedtuple` 对象的部分。这里保持数据的 RGB 三元组形式，不做进一步分解。

包含一组像素的图片可以表示为如下形式：

```
pixels = [(r, g, b), (r, g, b), (r, g, b), ...]
```

作为应用广泛的类库，PIL（Python Image Library）提供了多种像素呈现方式，包括从(x, y)形式的坐标值转换为 RGB 三元组。关于这个类库的更多信息，请参考 Pillow 项目文档（https://pypi.python.org/pypi/Pillow）。

对于一个给定的 `PIL Image` 对象，可使用如下脚本遍历其中每个元素：

```
from PIL import Image
from typing import Iterator, Tuple
Point = Tuple[int, int]
RGB = Tuple[int, int, int]
Pixel = Tuple[Point, RGB]
def pixel_iter(img: Image) -> Iterator[Pixel]:
    w, h = img.size
    return (
        (c, img.getpixel(c))
        for c in product(range(w), range(h))
    )
```

通过图像尺寸确定坐标范围，利用 `product(range(w), range(h))` 得到所有可能的像素坐标组合，实际上相当于两个嵌套的 `for` 循环。

这种处理方法的优点是每个像素都有各自的坐标位置，所以按照任意顺序处理像素都能还原整个图像，从而可以利用多进程技术或者多线程技术将计算负载分散到多个内核或者处理器上。Python 的 `concurrent.futures` 模块支持基于多核（处理器）的分布式计算。

9.2.1　计算距离

许多最优策略问题需要找到满足精度要求的近似解，即不能简单地判断值是否相等，而要采用某种距离测量算法，找到距离目标最近的元素。例如对于文本分析，可以采用 Levenshtein 距离，即从一段给定文本到目标文本需要做的变换次数的最小值。

下面举例说明，其中会涉及浅显的数学知识。虽然这个例子很简单，但要得到理想的结果仍应避免使用一些过于简单直接的算法。

处理颜色匹配问题通常不做精确的相等测试，因为很难检验像素颜色是否完全相同。我们往往先定义一个最短距离公式，然后计算两个颜色是否足够接近，但不要对像素的 R、G 和 B 都取同样的值。常用的距离计算公式包括欧氏距离、曼哈顿距离以及其他基于不同视觉偏好的复合加权计算公式。

欧氏距离和曼哈顿距离的计算公式如下：

```
import math
def euclidean(pixel: RGB, color: Color) -> float:
    return math.sqrt(
        sum(map(
            lambda x, y: (x-y)**2,
            pixel,
            color.rgb)
        )
    )

def manhattan(pixel: RGB, color: Color) -> float:
    return sum(map(
        lambda x, y: abs(x-y),
        pixel,
        color.rgb)
    )
```

欧氏距离表示 RGB 空间中某点与原点构成的直角三角形斜边的长度，**曼哈顿距离**则是直角三角形各边长度之和。欧氏距离精度高，曼哈顿距离计算速度快。

后续示例的数据结构比较类似，对每个特定的像素，计算其与给定颜色（属于一个由有限颜色组成的集合）的距离，结果如下所示：

```
(
 ((0, 0), (92, 139, 195), Color(rgb=(239, 222, 205), name='Almond'),
169.10943202553784),
 ((0, 0), (92, 139, 195), Color(rgb=(255, 255, 153), name='Canary'),
204.42357985320578),
 ((0, 0), (92, 139, 195), Color(rgb=(28, 172, 120), name='Green'),
103.97114984456024),
 ((0, 0), (92, 139, 195), Color(rgb=(48, 186, 143), name='Mountain Meadow'),
82.75868534480233),
 ((0, 0), (92, 139, 195), Color(rgb=(255, 73, 108), name='Radical Red'),
196.19887869200477),
 ((0, 0), (92, 139, 195), Color(rgb=(253, 94, 83), name='Sunset Orange'),
201.2212712413874),
 ((0, 0), (92, 139, 195), Color(rgb=(255, 174, 66), name='Yellow Orange'),
210.7961100210343)
)
```

计算结果是一个四元组集合，每个元素包含如下内容：

- 像素的坐标，例如`(0, 0)`；

- 像素的颜色，例如(92, 139, 195)；
- 7 种颜色集合中的某个 Color 对象，例如 Color(rgb=(239, 222, 205),name='Almond')；
- 像素颜色与给定 Color 对象间的欧氏距离。

不难发现欧氏距离的最小值就是最接近的匹配颜色，通过在归约中使用 min()函数可以方便地得到这个值。当把上面的颜色元组赋给 choices 变量时，像素级的归约实现如下：

```
min(choices, key=lambda xypcd: xypcd[3])
```

用 xypcd 代表每个四元组，也就是坐标、像素、颜色和距离。距离的最小值表示给定像素的最佳匹配颜色。

9.2.2　获得所有像素和颜色

如何获得包含所有像素和颜色的结构体？方法并不复杂，不过稍后你将看到，这种方法并不是最优解。

将像素映射到颜色的一种方法是使用 product()函数穷举所有像素和颜色。

```
xy = lambda xyp_c: xyp_c[0][0]
p = lambda xyp_c: xyp_c[0][1]
c = lambda xyp_c: xyp_c[1]

distances = (
    (xy(item), p(item), c(item), euclidean(p(item), c(item)))
    for item in product(pixel_iter(img), colors)
)
```

核心部分是使用 product(pixel_iter(img), colors)生成像素和颜色的所有组合，然后重构得到的数据使之扁平化，并使用 euclidean()函数算出像素颜色和 Color 对象颜色之间的距离。返回结果是一个四元组序列，元组元素分别为：*x*–*y* 坐标、源像素点、给定颜色，以及像素颜色到给定颜色间的距离。

最后使用 groupby()函数和 min(choices, ...)表达式得到颜色结果，如下所示：

```
for _, choices in groupby(
        distances, key=lambda xy_p_c_d: xy_p_c_d[0]):
    yield min(choices, key=lambda xypcd: xypcd[3])
```

像素和颜色做乘积运算得到一个很长的一维可迭代对象，按照其中的坐标值进行分组，将其分解为一组相对较短的可迭代对象，每个对应一个像素，然后选出距离最短的颜色。

在一幅包含 133 种 Crayola 色彩的、尺寸为 3648×2736 的图像中，上面的算法需要迭代计算 1 327 463 424 次。是的，distances 表达式生成了数十亿计的组合，这个规模还不至于无法计

算，Python 尚能处理，但足以体现简单直接地使用`produce()`函数会出现问题。

进行大规模数据处理时，必须估算规模。运行 100 万次距离计算后，用`timeit()`函数测出的运行时间如下：

- 欧氏距离：2.8
- 曼哈顿距离：1.8

扩大 1000 倍，从运行 100 万次到运行 10 亿次，曼哈顿距离计算需要 1800 秒，即半小时，欧氏距离计算则需要 46 分钟。对于大型数据集，这种计算方法效率太低了。

更重要的是，这种做法是错误的。这种“宽度×高度×颜色”的直接处理是糟糕的算法设计，很多情况下有更好的方案。

9.2.3　性能分析

大型数据算法的核心要素之一是执行某种分治策略，这点对函数式编程和命令式编程都是成立的。

可以通过下面 3 种方法提高处理速度。

- 使用并行策略实现并发计算，例如使用 4 核处理器，处理时间大约变为原来的 1/4，上面的曼哈顿距离计算可以缩短到 8 分钟左右。
- 保存中间计算结果以避免重复计算，搞清楚需要计算的相同颜色和不同颜色的数量。
- 使用新算法。

可以将后两种方法结合，比较所有源色彩和目标色彩。与逐个像素的计算方法相比，这种方式预先穷举所有可能的映射关系来避免重复计算。本质上这是将一系列比较运算转换成了在映射对象中进行查表操作。

对一张具体的图片进行预先的、源到目标的穷举映射计算时，首先要整体评估问题规模，本书随附的代码中包含一张名为“IMG_2705.jpg”的图片。下面的算法初步估算该图片的颜色元组的总规模：

```
from collections import defaultdict, Counter
palette = defaultdict(list)
for xy, rgb in pixel_iter(img):
    palette[rgb].append(xy)

w, h = img.size
print("Total pixels", w * h)
print("Total colors", len(palette))
```

对所有像素，按照其颜色值进行分组，将相同颜色的像素放入同一个列表中，从计算结果中

9

可以得到如下信息：

- 像素总数为 9 980 928，对于一个 10MB 大小的图片来说，这个数字在意料之中；
- 共有 210 303 种颜色，计算这些颜色与 133 种目标颜色之间的欧氏距离，只需 27 970 299 次计算，大约花费 76 秒；
- 使用 3 比特遮罩 `0b11100000` 时，在总共 $2^3 \times 2^3 \times 2^3$=512 种可能的颜色集合中，实际用到 214 种；
- 使用 4 比特遮罩 `0b11110000` 时，实际用到 1150 种；
- 使用 5 比特遮罩 `0b11111000` 时，实际用到 5845 种；
- 使用 6 比特遮罩 `0b11111100` 时，在总共 $2^6 \times 2^6 \times 2^6$ = 262 144 种颜色集合中，实际用到 27 726 种。

这有助于我们理解如何重组数据结构、快速实现颜色适配计算并重建图像，以避免进行十亿级的比较运算。

使用遮罩的作用是保留作用大的比特，去掉作用小的比特，比如有一个红色值为 200 的颜色，可以使用 Python `bin()`函数查看其二进制表示。

```
>>> bin(200)
'0b11001000'
>>> 200 & 0b11100000
192
>>> bin(192)
'0b11000000'
```

表达式 `200 & 0b11100000` 擦除了颜色的最低 5 比特，保留了最高 3 比特，最终结果是大小为 192 的红色值。

对 RGB 三元组的遮罩处理如下所示：

```
masked_color = tuple(map(lambda x: x & 0b11100000, c))
```

使用 `&` 运算符选择特定的比特，只保留红、绿、蓝的前 3 比特代替之前完整的颜色值，基于它们创建 `Counter` 对象后，可知使用遮罩后共产生 214 个颜色值，不到理论颜色值数量的一半。

9.2.4　重构问题

直接用 `produce()`函数比较所有像素和颜色显然不可取，毕竟 1000 万个像素只有 20 万种颜色。要把源色彩映射到目标色彩，只要将这 20 万种颜色保存在一个表中即可。

具体实现方法如下。

(1) 计算源色彩到目标色彩的映射，这里使用 3 比特表示输出值。R、G、B 三个量中，每个

的8种可能取值用`range(0, 256, 32)`表示。枚举所有输出值的表达式如下：`product(range(0, 256, 32), range(0, 256, 32), range(0, 256, 32))`

(2) 然后计算源色彩表中与最近颜色的欧氏距离，共需 68 096 次计算，耗时约 0.14 秒。这个步骤只需执行一次，生成 20 万个映射。

(3) 对图像做一次性处理，基于修正后的颜色表生成新图像。这里利用整数截断方法，通过类似于`(0b11100000 & r, 0b11100000 & g, 0b11100000 & b)`这样的表达式去掉颜色中不太重要的比特，后面的计算中还会用到该方法。

这样就将一个十亿级的欧氏距离计算转化为千万级的查表运算，耗时从30分钟降至30秒左右。

采用生成从源色彩到目标色彩的静态映射表，而不是对每个像素进行颜色映射，就建立了从原始颜色到新颜色的简单映射关系。

有了这个包含 20 万种颜色的色彩表，就可以通过它计算曼哈顿距离，确定相对于某种输出（例如蜡笔色彩）的最接近颜色了。用前面讲过的色彩匹配算法计算映射关系而不是结果图像，最主要的区别是用`palette.keys()`函数代替`pixel_iter()`函数。

稍后会介绍另一项优化技术——截断，用以加速算法。

9.2.5　合并两种变换

组合多个变换可以构建出复杂的映射关系，把原始数据转换为中间值，再生成最终结果。下面介绍如何将颜色截断和映射组合在一起。

在某些问题域中，难以执行截断操作，但在另一些场景中却很简单。例如从 9 位长的美国邮政编码中截取 5 位，或者进一步截取 3 位，来确定此邮编所在的大体地理方位。

对于颜色，可以使用前面讲的比特遮罩技术，将 3 个 8 比特值（共 24 比特，1600 万种颜色）转换为 3 个 3 比特值（共 9 比特，512 种颜色）。

创建一组颜色映射，其中包含到一组给定颜色的距离，以及对源颜色的截断，如下所示：

```
bit3 = range(0, 256, 0b100000)
best = (min((euclidean(rgb, c), rgb, c) for c in colors)
for rgb in product(bit3, bit3, bit3))
    color_map = dict((b[1], b[2].rgb) for b in best)
```

通过`range`创建对象`bit3`，遍历 3 比特颜色的所有 8 个值。使用二进制值`0b100000`有助于我们更好地理解使用比特的方法，忽略不重要的 5 比特，只使用最高的 3 比特。



`range`对象与普通可迭代对象不同，可以重复使用，所以这里用`product(bit3, bit3, bit3)`表达式生成所有 512 种颜色组合作为输出颜色。

我们为每个被截断的 RGB 颜色创建了一个三元组，包含元素：(0)到所有蜡笔颜色的距离，(1)这个 RGB 颜色本身，(2)代表蜡笔颜色的 `Color` 对象。对该集合求最小值，就得到了与这个被截断 RGB 颜色最接近的蜡笔颜色对象。

这样就有了从任何被截断 RGB 颜色到与它最接近的蜡笔颜色的映射关系对象。在查找一个与源色彩最匹配的蜡笔颜色前，首先要对源色彩做截断处理。这里的截断计算与预先计算得到的颜色映射表是对映射技术的组合使用。

实现图像替换的代码如下所示：

```
mask = 0b11100000
clone = img.copy()
for xy, rgb in pixel_iter(img):
    r, g, b = rgb
    repl = color_map[(mask&r, mask&g, mask&b)]
    clone.putpixel(xy, repl.rgb)
clone.show()
```

使用 PIL 库中的 `putpixel()`函数替换了图片中的像素。遮罩值保留了原始颜色值的最高 3 个比特，最终生成的颜色是原来颜色集合的一个子集。

可见使用函数式编程工具可以生成简洁明了但低效的算法，所以衡量算法复杂度的主要工具（也称**大 O 分析**）对于函数式编程和命令式编程同样重要。

这里的关键问题不是 `product()`函数低效，而是使用 `product()`函数创建的算法低效。

9.3　排列集合元素

排列集合元素指列出集合中元素的所有排列形式。对于长度为 n 的集合，共有 $n!$种排列形式，很多优化问题都使用排列进行暴力求解。

阅读维基百科词条 Combinatorial optimization，可知对于大规模问题，穷举所有排列并不适合，但对于小规模问题，使用 `itertools.permutations()`函数求解很方便。

组合优化的一个经典应用是分配任务：n 位员工要完成 n 项任务，但每个人完成某项任务的成本是不同的。同一项任务，某些员工处理起来成本很高，另一些员工处理起来成本却比较低。我们的任务是恰当地为每位员工分配任务，使得总成本最低。

不妨创建一个表格来记录每位员工完成每项任务的成本。假设现有 6 位员工要完成 6 项任务，则表格中要记录 36 个成本值，每格中的数字分别表示 0 到 5 号员工完成任务 A 到任务 F 需要的成本。

要列出集合的所有可能排序不难，但这种方法的扩展性不好。例如 10!等于 3 628 800，可以

用 list(permutations(range(10)))得到全部 300 多万个序列。

通常我们希望在几秒内就得到问题的答案。如果前面的问题规模翻一倍，到 20!，即 2 432 902 008 176 640 000 组排列时，就会出现扩展性问题。如果生成 10!个排列需要 0.56 秒，那么生成 20!组排列值则需要 12 000 年。

假设已有的成本矩阵包含 36 个值，显示 6 位员工完成 6 个任务的所有可能成本，可如下所示计算最优解：

```
perms = permutations(range(6))
alternatives = [
    (
        sum(
            cost[x][y] for y, x in enumerate(perm)
        ),
        perm
    )
    for perm in perms
]
m = min(alternatives)[0]
print([ans for s, ans in alternatives if s == m])
```

首先生成每名员工执行每个任务的所有排列，保存在 perms 中，然后创建每个组合与该组合的总成本的二元组。为了计算某项任务的成本，需要枚举各个组合，形成员工与任务二元组。例如有个组合是(1, 3, 4, 2, 0, 5)，对应的员工与任务二元组 list(enumerate((1, 3, 4, 2, 0, 5)))的值为[(0, 1), (1, 3), (2, 4), (3, 2), (4, 0), (5, 5)]。成本矩阵中组合值之和就是这项任务组合的总成本。

最小值对应的就是最优解。很多时候最优解不止一个，上面的算法可以找到所有最优解。表达式 min(alternatives)[0]返回最小值集合的第一个元素。

对于示例规模的问题，这种解法速度很快，但对于更大规模的问题，采用近似算法更适合。

9.4 生成所有组合

除了排列，itertools 模块还提供了计算集合元素组合的函数。对于组合来说，顺序不重要。对于一个给定的集合，组合的数量远小于排列的数量，对于 p 个元素组成的集合，r 元组合的数量为 $\binom{p}{r} = \frac{p!}{r!(p-r)!}$ 。

例如，5 张扑克牌共有 2 598 960 种组合方式，以下代码列出了所有组合形式：

```
hands = list(
    combinations(tuple(product(range(13), '♠♥♦♣')), 5))
```

实际应用中，在对包含多个变量的数据集进行探索性分析时，经常要计算任意两个变量间的相关性。如果有 v 个变量，可用下面的表达式枚举所有需要比较的变量对：

```
combinations(range(v), 2)
```

下面从 http://www.tylervigen.com 取样本数据来展示完整的处理流程。首先从中选择 3 个有共同时间范围的样本：第 7 号、第 43 号和第 3890 号，把它们放在同一个数据表中，保留各自的“年份”列。

数据表第一行和后面按年份排列的数据行如下所示：

```
[('year', 'Per capita consumption of cheese (US)Pounds (USDA)',
 'Number of people who died by becoming tangled in their bedsheetsDeaths (US) (CDC)',
 'year', 'Per capita consumption of mozzarella cheese (US)Pounds (USDA)',
 'Civil engineering doctorates awarded (US)Degrees awarded (National Science Foundation)',
 'year', 'US crude oil imports from VenezuelaMillions of barrels (Dept. of Energy)',
 'Per capita consumption of high fructose corn syrup (US)Pounds (USDA)'),
     (2000, 29.8, 327, 2000, 9.3, 480, 2000, 446, 62.6),
 (2001, 30.1, 456, 2001, 9.7, 501, 2001, 471, 62.5),
 (2002, 30.5, 509, 2002, 9.7, 540, 2002, 438, 62.8),
 (2003, 30.6, 497, 2003, 9.7, 552, 2003, 436, 60.9),
 (2004, 31.3, 596, 2004, 9.9, 547, 2004, 473, 59.8),
 (2005, 31.7, 573, 2005, 10.2, 622, 2005, 449, 59.1),
 (2006, 32.6, 661, 2006, 10.5, 655, 2006, 416, 58.2),
 (2007, 33.1, 741, 2007, 11, 701, 2007, 420, 56.1),
 (2008, 32.7, 809, 2008, 10.6, 712, 2008, 381, 53),
 (2009, 32.8, 717, 2009, 10.6, 708, 2009, 352, 50.1)]
```

使用 `combinations()` 函数基于 9 个变量生成所有二元比较对。

```
combinations(range(9), 2)
```

共有 36 种组合，去掉其中由各个年份列形成的组合，它们的相关系数是 1.00。

从数据集中提取列的函数如下所示：

```
from typing import TypeVar, Iterator, Iterable
T_ = TypeVar("T_")
def column(source: Iterable[List[T_]], x: int) -> Iterator[T_]:
    for row in source:
        yield row[x]
```

然后用第 4 章中的 `corr()` 函数比较两列数据。

如下所示计算所有组合相关系数：

```
from itertools import *
from Chapter_4.ch04_ex4 import corr
for p, q in combinations(range(9), 2):
    header_p, *data_p = list(column(source, p))
    header_q, *data_q = list(column(source, q))
```

```
    if header_p == header_q:
        continue
    r_pq = corr(data_p, data_q)
    print("{2: 4.2f}: {0} vs {1}".format(header_p, header_q, r_pq))
```

对于组合在一起的列，首先将它们从数据集中提取出来，header_p, *data_p =语句通过多重赋值将序列的第一个值（即标题）与后面的数据分离。如果标题一致，说明参与计算的是同一列。在上面的数据集中，由于存在3个重复的年份列，所以要排除这种情况。

之后用相关性函数处理这些列，得到相关系数，再打印出这些列的标题，这里特意选择了几个模式不同但相关度很高的伪相关特征。

计算结果如下：

```
0.96: year vs Per capita consumption of cheese (US) Pounds (USDA)
0.95: year vs Number of people who died by becoming tangled in their
bedsheetsDeaths (US) (CDC)
0.92: year vs Per capita consumption of mozzarella cheese (US) Pounds (USDA)
0.98: year vs Civil engineering doctorates awarded (US) Degrees awarded (National
Science Foundation)
-0.80: year vs US crude oil imports from Venezuela Millions of barrels
(Dept. of Energy)
-0.95: year vs Per capita consumption of high fructose corn syrup (US) Pounds (USDA)
0.95: Per capita consumption of cheese (US) Pounds (USDA) vs Number of people who
died by becoming tangled in their bedsheetsDeaths (US) (CDC)
0.96: Per capita consumption of cheese (US) Pounds (USDA) vs year
0.98: Per capita consumption of cheese (US) Pounds (USDA) vs Per capita
consumption of mozzarella cheese (US) Pounds (USDA)
...
0.88: US crude oil imports from VenezuelaMillions of barrels (Dept. of Energy)
vs Per capita consumption of high fructose corn syrup (US) Pounds (USDA)
```

数据体现出的模式的意义尚不清楚，为什么存在相关性？这些缺乏明确意义的、含混的相关性会干扰统计分析，但我们找到了那些相关性很高却缺乏关联因素的数据。

这里的重点是使用简单的表达式 combinations(range(9), 2)生成了所有可能的数据组合。利用这类简单易用的技术让我们可以专注于处理数据分析中的问题，而不必费心于构建组合算法。

9.5　代码范例

Python 库文档中 itertools 章的 Itertools Recipes 部分很精彩，基本定义后面跟着一系列范例，逻辑清晰且实用性强。这里不会重复文档中的内容，而是直接给出出处。学习 Python 函数式编程，这部分文档属于必读内容。

Python 标准库文档 10.1.2 节“Itertools Recipes”是非常好的学习资源，见 https://docs.python.

org/3/library/itertools.html#itertools-recipes。

需要说明的是，这里列出的范例都不是 itertools 模块中可以导入的函数，如果想在自己开发的应用中使用这些范例，需要阅读代码并理解原理，然后通过复制、修改来使用代码。

下面的表格总结了基于 itertools 模块中的基本函数实现的一些函数式编程范例。

函　数　名	参数列表	返回结果
powerset	(iterable)	生成输入参数 iterable 的所有子集，每个子集是一个元组对象，不是集合实例
random_product	(*args, repeat=1)	从 itertools.product(*args, **kwds)随机返回部分值
random_permutation	(iterable, r=None)	从 itertools.permutations(iterable, r)随机返回部分值
random_combination	(iterable, r=None)	从 itertools.combinations(iterable, r)随机返回部分值

9.6　小结

本章重点介绍了 itertools 模块中的几个函数，利用该组件库提供的工具，开发者可以更好地使用可迭代对象。

product()函数从两个或多个集合中选择元素，返回所有可能的组合，permutations()函数返回输入集合的所有可能排列，而 combinations()函数返回输入集合的所有子集。

本章还介绍了如何以简单直接的方式使用 product()和 permutations()函数来生成巨大的结果数据集，旨在提醒开发者：即便在简单明了的算法实现中，也可能涉及巨大的计算量，开发者需要预估复杂度，确保算法能在可接受的时间内给出结果。

下一章将详细介绍 functools 模块，其中一些工具用于处理作为头等对象的函数，部分实现基于第 2 章和第 5 章的内容。

第 10 章

functools 模块

函数式编程强调将函数作为头等对象。前面介绍过用函数作为参数和返回值的高阶函数，本章将介绍 functools 库中用于创建和修改函数的几个高阶函数。

本章主要介绍高阶函数，第 5 章讲过它，第 11 章还会介绍高阶函数相关技术。

本章将介绍以下函数。

- @lru_cache：该装饰器可以显著提升某些应用的性能。
- @total_ordering：该装饰器有助于创建功能强大的比较运算符，促使我们认真思考面向对象编程和函数式编程混合使用时需要解决的一些普遍性问题。
- partial()：可以利用该函数将部分参数传给指定函数，从而创建一个新函数。
- reduce()：创建抽象规约（例如 sum()）的高阶函数。

functools 库中的两个函数 update_wrapper() 和 wraps() 留待下一章详述，下一章还会介绍如何编写自定义装饰器。

本章不讨论 cmp_to_key() 函数，它让基于比较的 Python 2 版本的代码能在基于键值提取的 Python 3 中运行。本书立足于 Python 3，只关注如何正确编写键值函数。

10.1 函数工具

第 5 章介绍过几个高阶函数，它们以函数为参数或者返回函数（以及生成器表达式）。可以通过注入一个外部函数来定制这些高阶函数定义的算法，max()、min()、sorted() 等函数通过接收一个 key= 函数实现定制，map()、filter() 等函数则通过接收一个函数和一个可迭代对象来将函数参数应用于可迭代对象。map() 函数对计算结果做 yield 处理，filter() 函数利用参数函数返回的布尔值决定对可迭代对象中值的取舍。

第 5 章介绍的所有函数都在 Python 的 __builtins__ 包内，无须使用 import 语句即可使用。这些函数应用非常广泛，无须专门导入。本章介绍的函数则必须通过 import 语句导入，因为它们不如前者应用广泛。

这里有个特殊情况：reduce()函数原本内置在__builtins__包中，后来为了避免滥用，经过社区讨论，将它移出了__builtins__包。使用该函数执行一些似乎很简单的操作时，也会出现明显的性能问题。

10.2　使用 lru_cache 保存已有计算结果

使用@lru_cache 装饰器能加快函数运行，其中 LRU 指最近使用的（least recent used）——最近使用过的计算结果会保留在这个容量有限的池子中。为了避免池中内容无限制地增加，会清除不常用的计算结果。

作为装饰器，可以将它应用于需要保存之前计算结果的任何函数，示例如下：

```
from functools import lru_cache
@lru_cache(128)
def fibc(n: int) -> int:
    if n == 0: return 0
    if n == 1: return 1
    return fibc(n - 1) + fibc(n - 2)
```

该示例基于第 6 章的斐波那契数列算法实现。由于对简单的斐波那契数列算法应用了@lru_cache 装饰器，每次调用 fibc(n)时，都会检查由装饰器维护的缓存池。如果参数 n 在缓存池中，直接使用对应的计算结果，避免了重复计算导致的成本增加。每次计算的结果都会保存到缓存池中。

上例意在展示斐波那契数列的简单递归实现的计算量是非常大的，每次计算给定的斐波那契数 F_n时，不仅要计算 F_{n-1}，而且要计算 F_{n-2}，这棵递归树的计算复杂度是 $O(2^n)$。

装饰器中的参数 128 定义了缓存池的大小，用于限制内存的使用上限。当缓存池满时，LRU 计算结果会被新值覆盖。

下面通过 timeit 模块验证该装饰器提升性能的幅度。将两个实现各运行 1000 次，比较所用的时间。分别运行 fib(20)和 fibc(20)，就可以看到不用缓存的计算极慢，由于计算耗费的时间过多，这里把 timeit 重复计算的次数仅设置为 1000，计算消耗的时间如下所示。

- 非缓存实现：3.23
- 带缓存实现：0.0779

请注意，不能将 timeit 模块直接应用于函数 fibc()，由于缓存的存在，直接应用时只有一次真正计算，计算结果保存在缓存池中，其他 999 次计算直接从缓存中取结果，并没有再次计算。每次计算后需要清空缓存，否则总计算时间将接近 0。清空计算结果由装饰器的 fibc.cache_clear()方法实现。

缓存是个很强大的工具，很多算法都可以通过缓存结果提升性能。

在包含 p 个元素的集合中抽取 r 个元素的公式如下：

$$\binom{p}{r}=\frac{p!}{r!(p-r)!}$$

该二项分布函数包含了 3 个阶乘计算。在阶乘函数上使用`@lru_cache`很有必要，借助缓存计算二项分布值就不必重复计算阶乘了。如果计算过程包含大量重复计算，使用缓存可以显著缩短计算时间，但在计算结果很少被复用的场景中，维护缓存的开销可能会抵消速度提升。

对于上面这个包含大量重复计算的场景，有无缓存消耗的时间分别如下。

- 无缓存阶乘：0.174
- 有缓存阶乘：0.046

需要指出的是，缓存是有状态对象，基于缓存的实现方案不符合纯函数式编程要求。理想的函数式编程应尽量避免状态变化，例如函数式编程中多使用递归取代值有状态的变量，当前状态由函数的参数定义，而不是保存在值可变的变量里。前面讲过运用尾递归优化技术，即使只用性能有限的处理器和内存，也能显著提升性能。在 Python 中，我们使用 `for` 循环手动实现尾递归优化。缓存是一种类似的优化技术，如有需要便手动实现它，但它本身不是纯函数式编程。

原则上调用带 LRU 的函数有两个结果：返回本次计算结果，并将其保存到缓存池中供后续使用。被缓存的值封装在带装饰器的`fibc()`函数内部，无法查看或操作。

缓存技术不是灵丹妙药，涉及大量实数计算的应用中，由于实数的近似效应，缓存的效果往往不好，往往将实数的最小有效数字视作随机噪声，导致`@lru_cache`装饰器的目标查找难以正常工作。

第 16 章将介绍解决这一问题的方法。

10.3 使用 total_ordering 定义类

`total_ordering`装饰器用于定义实现各种比较运算的算子类，既可用于`numbers.Number`的子类，也可用于半数值型类。

下面以扑克牌为例说明使用半数值型类的场景。扑克牌有点数，也有花色，有些玩法中点数起的作用非常大。与数字类似，扑克牌可以排序，点数也可以相加。但牌与牌之间做乘法毫无意义，这一点又与数字不同。

可以通过继承 `NamedTuple` 类定义一个扑克牌类，如下所示：

```
from typing import NamedTuple
class Card1 (NamedTuple):
    rank: int
    suit: str
```

该定义看上去不错，但有个问题：所有实例的比较都必须包含点数和花色，因此当比较黑桃 2 和梅花 2 时会出现下面的情况：

```
>>> c2s= Card1(2, '\u2660')
>>> c2h= Card1(2, '\u2665')
>>> c2s
Card1(rank=2, suit='♣')
>>> c2h= Card1(2, '\u2665')
>>> c2h
Card1(rank=2, suit='♥')
>>> c2h == c2s
False
```

不难发现这个类的默认比较方式不适合许多玩法。

大多数扑克牌玩法都只需要比较点数，更实用的定义如下所示：

```
from functools import total_ordering
from numbers import Number
from typing import NamedTuple

@total_ordering
class Card2(NamedTuple):
    rank: int
    suit: str
    def __eq__(self, other: Any) -> bool:
        if isinstance(other, Card2):
            return self.rank == other.rank
        elif isinstance(other, int):
            return self.rank == other
        return NotImplemented
    def __lt__(self, other: Any) -> bool:
        if isinstance(other, Card2):
            return self.rank < other.rank
        elif isinstance(other, int):
            return self.rank < other
        return NotImplemented
```

这里的 Card2 类继承了 NamedTuple 类，使用父类的__str__()方法将实例以字符串的形式打印出来。

类中定义了两个比较方法：一个定义相等，一个定义顺序。其他比较方法由@total_ordering 基于这两个定义完成，包括__le__()、__gt__()和__ge__()等，不等比较方法__ne__()默认基于__eq__()生成，装饰器无须参与。

以上方法实现了两种比较：两个 Card2 对象之间，以及 Card2 对象和整形数值之间。__eq__()和__lt__()参数的类型标示必须是 Any，以保证与父类兼容，虽然写成 Union[Card2, int] 更精确，但会与父类冲突。

首先这个类提供了仅基于点数的比较，如下所示：

```
>>> c2s= Card2(2, '\u2660')
>>> c2h= Card2(2, '\u2665')
>>> c2h == c2s
True
>>> c2h == 2
True
>>> 2 == c2h
True
```

这个类可用于扑克牌之间基于点数的比较，装饰器自动生成了多种比较运算符，如下所示：

```
>>> c2s= Card2(2, '\u2660')
>>> c3h= Card2(3, '\u2665')
>>> c4c= Card2(4, '\u2663')
>>> c2s <= c3h < c4c
True
>>> c3h >= c3h
True
>>> c3h > c2s
True
>>> c4c != c2s
True
```

这里无须手动编写比较运算符，装饰器会自动生成，但它生成的运算符并不是完全理想的。对于本例，如果要比较整数和 `Card2` 对象，就出现了问题。

由于运算符解析机制的限制，类似于 `c4c > 3` 和 `3 < c4c` 这样的操作会出现 `TypeError` 异常，这是 `total_ordering` 无法正确处理的情形。虽然在实际应用中这样的情况不常见，但当确实需要这样的比较时，所有的比较运算符都要手动定义，而不能使用`@total_ordering` 装饰器自动生成。

面向对象编程并不是函数式编程的对立面，很多时候二者是互补的。Python 生成不可变对象的能力恰好契合了函数式编程的要求。我们完全可以既避免创建带有复杂状态的对象，又能将相关方法封装在一起使用。尤其当类属性中包含复杂计算时，将复杂计算封装在类定义中会让应用的逻辑更易理解。

定义数字类

有时需要拓展 Python 已有的数值系统，通过继承 `numbers.Number` 类来简化函数式编程。例如可以把复杂的算法封装在 `Number` 的子类中，以简化或者明晰化应用的其他部分。

Python 提供了丰富的数值类型，内置的 `int` 类型和 `float` 类型覆盖了大多数应用场景。使用 `decimal.Decimal` 包能很好地处理货币相关问题。某些场景中，`fractions.Fraction` 比 `float` 更适用。

例如处理地理数据，可以通过继承 `float` 类，引入新的属性来实现经度或者纬度与弧度之间的转换。利用这个类可以方便地处理跨越赤道或者格林威治子午线时需要进行的计算（$\mathrm{mod}(2\pi)$）。

由于 Python 的 Number 类是不可变的，常用的函数设计方法都可以在这里使用，对于少数自身状态会变化的函数（例如__iadd__()函数），直接忽略即可。

使用 Number 的子类时，需要考虑下面几点。

- 相等测试与哈希值计算。哈希值计算的核心特点可参考 Python 标准库文档 9.1.2 节。
- 其他比较运算符（通常由@total_ordering 装饰器定义）。
- 算术运算符：+、-、*、/、//、%和**，包括处理前置运算符的特殊方法和反向类匹配的附加方法。对于表达式 a-b，Python 尝试在 a 的类型中寻找实现__sub__()方法的函数，即 a.__sub__(b)方法。如果左侧变量（这里是 a）的类没有这个方法，或者出现 NotImplemented 异常，再检查右侧变量是否提供了 b.__rsub__(a)方法。当 b 是 a 的子类时为特殊情况，允许子类的实现方法覆盖左侧变量的实现方法。
- 位运算算子：&、|、^、>>、<<和~。实数无须考虑这些运算符，直接忽略即可。
- 其他函数如 round()、pow()、divmod()等由专门的数值处理方法实现，对这里定义的类可能有意义。

第 7 章有个例子详细演示了创建新数值类型的过程，具体过程可参考 https://www.packtpub.com/application-development/mastering-object-oriented-python。

前面讲过，函数式编程和面向对象编程是互补的，完全可以按照函数式编程的要求定义类。创建数值类型的例子很好地展示了如何利用 Python 的面向对象特征创建易读的函数式程序。

10.4　使用 partial()函数应用部分参数

partial()函数生成的是所谓的“部分应用”。部分应用的函数是基于旧函数及其部分参数生成的新函数，与柯里化密切相关。由于 Python 函数不是通过柯里化实现的，因此这里不介绍其理论背景。不过这个概念有助于我们理解简化。

示例如下：

```
>>> exp2 = partial(pow, 2)
>>> exp2(12)
4096
>>> exp(17)-1
131071
```

这里创建了函数 exp2(y)，实际上是 pow(2, y)。partial()函数将 pow()函数的第一个位置参数与 pow()函数绑定，当对新生成的 exp2()函数求值时，通过 partial()绑定的参数和 exp2()自己的参数参与了求值。

位置参数的绑定是按照从左到右的严格顺序完成的。如果函数接收关键字参数，也按相同的规则进行。

部分应用函数也可以通过匿名函数实现，如下所示：

```
exp2 = lambda y: pow(2, y)
```

两种实现的效果完全一致。性能测试表明 `partial()`函数比匿名函数实现的方式略快：

- ❑ `partial()`: 0.37
- ❑ 匿名函数：0.42

100 万次循环运行快 0.05 秒，优势不明显。

由于匿名函数方式能实现 `partial()`函数的所有功能，暂不详述。第 14 章会研究如何通过柯里化达到相同的目的。

10.5 使用 `reduce()`函数归约数据集

可以将 `sum()`、`len()`、`max()`和 `min()`函数看作 `reduce()`函数的特殊形式。`reduce()`函数是高阶函数，能将可迭代对象中相邻的两个值通过指定函数结合在一起。

现有如下序列对象：

```
d = [2, 4, 4, 4, 5, 5, 7, 9]
```

`reduce()`函数对它应用+运算符后效果如下：

```
2 + 4 + 4 + 4 + 5 + 5 + 7 + 9
```

为了更好地说明计算过程，给上面的表达式加上括号。

```
((((((2 + 4) + 4) + 4) + 5) +5 ) + 7) + 9
```

Python 对表达式的标准解释方式是从左到右求值，所以与左卷积（fold-left）是同一个意思。有些函数式语言提供了右卷积（fold-right）函数，当与递归组合使用时，这些函数会进行优化处理。不过在 Python 中，归约都是从左到右进行的，所以不存在优化问题。

可以给归约提供一个初始值，如下所示：

```
reduce(lambda x, y: x + y ** 2, iterable, 0)
```

如果不提供，则将序列的第一个值用作初始值。设置初始值的方式对于 `map()`函数和 `reduce()`函数都非常重要。下面通过设置初始值为 0 得到正确的计算结果：

```
0 + 2**2 + 4**2 + 4**2 + 4**2 + 5**2 + 5**2 + 7**2 + 9**2
```

如果不设置初始值，`reduce()`函数使用序列的第一个值作为初始值，这个值就不会传递给卷积函数，导致计算错误。没有初始值的 `reduce()`函数的计算过程如下所示：

10

```
2 + 4**2 + 4**2 + 4**2 + 5**2 + 5**2 + 7**2 + 9**2
```

这样的错误告诉我们使用 reduce()函数时务必小心。

下面通过 reduce()高阶函数定义一些内置的归约函数。

```
sum2 = lambda data: reduce(lambda x, y: x + y ** 2, data, 0)
sum = lambda data: reduce(lambda x, y: x + y, data, 0)
count = lambda data: reduce(lambda x, y: x + 1, data, 0)
min = lambda data: reduce(lambda x, y: x if x < y else y, data)
max = lambda data: reduce(lambda x, y: x if x > y else y, data)
```

其中 sum2()归约函数计算序列的平方和，在求样本集的标准差时很有用。sum()归约函数模仿内置的 sum()函数的功能。count()归约函数与 len()函数的功能类似，不过前者可以接收可迭代对象作为参数，而后者只能处理实例化的集合对象。

min()函数和 max()函数模仿内置同名函数的功能。这里将序列第一个值作为初始值以保证计算结果正确。如果对 reduce()函数指定了额外的初始值，则会由于提供了一个序列中不存在的值而得到错误的结果。

10.5.1　合并 map()和 reduce()

不难发现基于一些简单的定义就可以创建高阶函数。下面介绍如何组合 map()函数和 reduce()函数生成 map-reduce 函数。

```
from typing import Callable, Iterable, Any
def map_reduce(
        map_fun: Callable,
        reduce_fun: Callable,
        source: Iterable) -> Any:
    return reduce(reduce_fun, map(map_fun, source))
```

这个由 map()函数和 reduce()函数组成的函数有 3 个参数：一个映射操作，一个归约操作，以及一个待处理的数据序列。

上面的定义非常宽泛，很难从中看出对数据类型的要求。由于映射和归约可能是非常复杂的操作，下面更严格地定义 map-reduce 函数：

```
from typing import Callable, Iterable, TypeVar
T_ = TypeVar("T_")
def map_reduce(
        map_fun: Callable[[T_], T_],
        reduce_fun: Callable[[T_, T_], T_],
        source: Iterable[T_]) -> T_:
    return reduce(reduce_fun, map(map_fun, source))
```

该定义多了几项约束。首先，可迭代对象需要包含类型一致的数据，我们把这个类型绑定为 T_类型变量；然后，map()函数接收类型为 T_的参数，并生成相同类型的结果；最后，归约函

数接收两个 T_类型的参数，返回一个同类型的参数。对于一个简单的数值计算应用来说，使用类型变量 T_施加类型约束效果很好。

更多时候需要更严格地定义映射函数，例如 Callable[[T1_], T2_]就体现了映射函数的核心特点：输入类型 T1_和输出类型 T2_可能是不同的。归约函数则需要保证输入和输出函数类型一致：Callable[[T2_, T2_], T2_]。

可以分别提供映射函数和归约函数来得到计算平方和的归约定义，如下所示：

```
def sum2_mr(source: Iterable[float]) -> float:
    return map_reduce(
        lambda y: y ** 2, lambda x, y: x + y, source)
```

这里使用了 lambda y: y ** 2 作为平方计算的映射函数，归约部分使用 lambda x, y: x + y 作为参数。无须专门指定初始值，因为它就是平方函数映射后序列的第一个值。

上面的参数 lambda x, y: x + y 相当于+运算符，Python 的 operator 模块中将所有算术运算符定义为短函数，下面用它简化上面的 map-reduce 定义：

```
from operator import add
def sum2_mr2(source: Iterable[float]) -> float:
    return map_reduce(lambda y: y ** 2, add, iterable)
```

这里使用了 operator.add 方法代替匿名函数来对数据进行求和。

计算可迭代对象中数值的个数如下所示：

```
def count_mr(source: Iterable[float]) -> float:
    return map_reduce(lambda y: 1, lambda x, y: x+y, source)
```


首先通过 lambda y: 1 将每个元素映射为 1，再通过匿名函数或者 operator.add 做归约汇总，就得到了元素的个数。

总的说来，可以使用 reduce()函数创建任何类型的归约，将大型数据集转换为单个数值。不过在使用 reduce()函数时，需要注意一些限制。

避免像下面这样使用 reduce()函数：

```
reduce(operator.add, list_of_strings, "")
```

该函数能运行，Python 可以把 add 运算符应用于字符串集合，但使用"".join (list_of_strings)效率更高。通过 timit 测试发现使用 reduce()组合字符串的时间复杂度是 $O(n^2)$，且非常慢。在实际应用中，如果不仔细研究运算符处理复杂数据集的机制，很难发现效率的瓶颈在哪里。

10.5.2　使用 reduce()函数和 partial()函数

可以将 sum()函数定义为 partial(reduce, operator.add)，这也提示我们可以用类似的方法创建其他映射和归约。可以通过部分应用函数而不是匿名函数定义常用的归约函数，如下所示：

```
sum2 = partial(reduce, lambda x, y: x + y ** 2)
count = partial(reduce, lambda x, y: x + 1)
```

现在可以通过 sum2(some_data)或者 count(some_iter)来使用这些函数了。前面讲过，这样的实现方法性能优势不明显，但仍然不失为一项重要的技术，在处理特别复杂计算时可以使用部分应用函数实现简化。

10.5.3　使用 map()函数和 reduce()函数清洗数据

清洗数据时经常需要引入复杂度不同的过滤器来剔除无效数据，下面通过定义映射将有效但格式不规范的数据，转换为有效且格式规范的数据。

首先定义如下函数：

```
def comma_fix(data: str) -> float:
    try:
        return float(data)
    except ValueError:
        return float(data.replace(",", ""))

def clean_sum(
        cleaner: Callable[[str], float],
        data: Iterable[str]
    ) -> float:
    return reduce(operator.add, map(cleaner, data))
```

comma_fix()函数通过去掉字符串中的逗号，将格式不规范的表达数字的字符串转换为实数，类似的场景还有去掉字符串中的$符号并转换为 decimal.Decimal 类型数据。

然后可以将一个执行清洗操作的映射函数（这里是 comma_fix()类）应用于数据，再用 operator.add 方法进行归约。

清洗过程如下所示：

```
>>> d = ('1,196', '1,176', '1,269', '1,240', '1,307',
... '1,435', '1,601', '1,654', '1,803', '1,734')
>>> clean_sum(comma_fix, d)
14415.0
```

这样就通过修正逗号实现了清洗数据并计算总和，组合两项操作的语法也很简单。

请注意，应避免多次调用清洗函数，比如计算一组数据的平方和时，不应执行如下命令：

```
comma_fix_squared = lambda x: comma_fix(x) ** 2
```

由于计算数据的标准差时也要用到 `clean_sum(comma_fix, d)`方法，就执行了两次修正逗号的操作，一次计算数据总和，一次计算平方和，这样的算法设计很差。用 `lru_cache` 装饰器缓存计算结果会有帮助，更好的方法是将清洗的中间结果实例化到临时的元组对象中。

10.5.4 使用 groupby()函数和 reduce()函数

数据分析中经常需要分组数据并总结分组情况。可以用 `defaultdict(list)`进行分组，然后展开分析。第 4 章介绍了如何排序和分组，第 8 章介绍过其他方法。

需要处理的数据实例如下所示：

```
>>> data = [('4', 6.1), ('1', 4.0), ('2', 8.3), ('2', 6.5),
... ('1', 4.6), ('2', 6.8), ('3', 9.3), ('2', 7.8),
... ('2', 9.2), ('4', 5.6), ('3', 10.5), ('1', 5.8),
... ('4', 3.8), ('3', 8.1), ('3', 8.0), ('1', 6.9),
... ('3', 6.9), ('4', 6.2), ('1', 5.4), ('4', 5.8)]
```

原始数据序列中每个元素包含一个键值以及对键值的度量。

创建分组的一个常用方法是将键值相同的元素放入同一个列表，如下所示：

```
from collections import defaultdict
from typing import (
    Iterable, Callable, Dict, List, TypeVar,
    Iterator, Tuple, cast)
D_ = TypeVar("D_")
K_ = TypeVar("K_")

def partition (
        source: Iterable[D_],
        key: Callable[[D_], K_] = lambda x: cast(K_, x)
    ) -> Iterable[Tuple[K_, Iterator[D_]]]:
    pd: Dict[K_, List[D_]] = defaultdict(list)
    for item in source:
        pd[key(item)].append(item)
    for k in sorted(pd):
        yield k, iter(pd[k])
```

上述代码按照键值将可迭代对象中的数据分到不同的组中，其中将可迭代对象中源数据的数据类型定义为 `D_`，代表每个数据项。用 `key()`函数从数据项中抽取键值，将返回结果的类型定义为 `K_`，以与数据项的类型 `D_`做区分。从示例数据可以看出，每个数据项是一个元组（类型标识符为 `tuple`），每个键值是一个字符串（类型标识符为 `str`），键值抽取参数作为可调用函数，将元组类型转换为字符串类型。

key()函数从数据项中抽取的字符串作为了 pd 字典的键值，将数据项追加到了对应的列表中。defaultdict 对象将类型为 K_的键值映射到了类型为 List[D_]的数据列表上。

上述函数返回结果的数据结构与 itertools.groupby()函数的一致，都是由(group key, iterator)元组组成的可迭代序列，其中分组键值的类型由键值函数的返回值的类型决定，迭代器中则包含一系列原始数据项。

基于 itertools.groupby()实现的分组函数如下所示：

```
from itertools import groupby

def partition_s(
        source: Iterable[D_],
        key: Callable[[D_], K_] = lambda x: cast(K_, x)
    ) -> Iterable[Tuple[K_, Iterator[D_]]]:
    return groupby(sorted(source, key=key), key)
```



注意：上述两种实现的输入部分有个重要的差别：groupby()函数要求数据必须是按键值排序好的，defaultdict 则不需要事先排序。对于大型数据集，不论从时间上还是存储空间上排序，成本都会非常高。

接下来使用上面定义的函数对数据进行分组，可以把该操作作为分组过滤的预处理或分组统计的预处理。

```
>>> for key, group_iter in partition(data, key=lambda x:x[0]):
... print(key, tuple(group_iter))
1 (('1', 4.0), ('1', 4.6), ('1', 5.8), ('1', 6.9), ('1', 5.4))
2 (('2', 8.3), ('2', 6.5), ('2', 6.8), ('2', 7.8), ('2', 9.2))
3 (('3', 9.3), ('3', 10.5), ('3', 8.1), ('3', 8.0), ('3', 6.9))
4 (('4', 6.1), ('4', 5.6), ('4', 3.8), ('4', 6.2), ('4', 5.8))
```

对分组数据的统计汇总如下所示：

```
mean = lambda seq: sum(seq)/len(seq)
var = lambda mean, seq: sum((x-mean)**2/mean for x in seq)

Item = Tuple[K_, float]
def summarize(
        key_iter: Tuple[K_, Iterable[Item]]
    ) -> Tuple[K_, float, float]:
    key, item_iter = key_iter
    values = tuple(v for k, v in item_iter)
    m = mean(values)
    return key, m, var(m, values)
```

partition()函数的返回结果是一系列(key, iterator)二元组，summarize()函数接收这类二元组，从输入数据项中抽取键值和数据迭代器。上面的实现将数据项的类型定义为 Item，包含一个类型为 K_的键值和一个可以转换为浮点数的数值。为了从 item_iter 迭代器中抽取出数值部分，使用生成器表达式得到一个仅包含值的元组。

还可以使用(snd, item_iter)实现从二元组中抽取第二项，并且 snd = lambda x: x[1]。snd 这个名字是从元组中抽取第二项中“第二”（second）的简写。

将 summarize()函数应用于每个分组，如下所示：

```
>>> partition1 = partition(data, key=lambda x: x[0])
>>> groups1 = map(summarize, partition1)
```

或者使用另一个实现方案，如下所示：

```
>>> partition2 = partition_s(data, key=lambda x: x[0])
>>> groups2 = map(summarize, partition2)
```

两种方法都能计算出每个分组的统计汇总值，计算结果如下所示：

```
1 5.34 0.93
2 7.72 0.63
3 8.56 0.89
4 5.5 0.7
```

这里算出的方差可用于卡方检验，用于确定数据集的零假设是否成立。零假设指数据不包含有价值的信息：方差是完全随机的。还可以对数据做组间比较，确定各组平均值的变化是否符合零假设，或者存在某种统计意义上的显著变化。

10.6 小结

本章介绍了 functools 库中的几个函数，利用它们可以方便地创建功能强大的函数和类。

对于需要重复计算同一组值的应用，@lru_cache 函数可以大幅提升其性能。对于用 integer 或 string 作为参数的函数来说，该装饰器非常有用，它可以通过保存已有的计算结果来缩短后续处理时间。

@total_ordering 装饰器有助于实现复杂的排序比较功能，该函数似乎不属于函数式编程的重点，但在创建新类型数时非常有用。

partial()函数通过传递部分参数创建新函数，使用匿名函数也能实现类似的效果，两种方法可以互换。

最后介绍了高阶函数 reduce()，它是归约函数（例如 sum()）的一般化。后面章节的实例仍会用到这个函数，它与 filter()和 map()都是重要的高阶函数。

下一章将介绍如何通过装饰器创建高阶函数，如此创建的高阶函数语法更简单清晰。当函数需要与其他许多函数和类相互调用时，装饰器是个很有用的工具。

第 11 章

装饰器设计技术

Python 提供了许多创建高阶函数的方法。第 5 章介绍了两种技术：定义接收函数作为参数的函数，以及定义 `Callable` 的子类，后者可以通过函数初始化或作为参数被函数调用。

使用装饰器函数便于创建复合函数。复合函数是能表征多个源函数功能的单个函数。对于一个复杂算法，复合函数 $f \circ g(x)$比 $f(g(x))$更清晰明了。对于开发者来说，有多种语法备选方案可表示复杂运算，通常会很有帮助。

本章讨论以下几个主题：

- 使用装饰器构建基于其他函数的函数；
- `functools` 模块中的 `wraps()`函数，可用于构建装饰器；
- `update_wrapper()`函数亦有所用。

11.1　作为高阶函数的装饰器

装饰器的核心思想是将某些原始函数转换成另一种形式。装饰器基于装饰符和原始被装饰函数创建复合函数。

可以通过以下两种方式使用装饰器函数。

以前缀形式创建一个与基函数同名的新函数，如下所示：

```
@decorator
def original_function():
    pass
```

以显式运算的形式返回一个新函数（名字可能不同）。

```
def original_function():
    pass
original_function = decorator(original_function)
```

以上是针对同一操作的两种语法。前缀表示法的优点是简洁明了。装饰符在前缀位置对于一

些读者来说更易读。后缀表示法是显式的，且略微灵活一些。

尽管前缀表示法很常用，但有时会使用后缀表示法，因为我们不希望生成的函数取代原来的函数。我们更希望执行以下命令，以同时使用装饰后和装饰前的函数。

```
new_function = decorator(original_function)
```

这将由原始函数构建出一个名为 `new_function()`的新函数。Python 函数是头等对象，因此使用 `@decorator` 语法时，原始函数将不再可用。

装饰器是能接收函数作为参数并且返回函数的函数，这简单描述了 Python 语言的内置特性，但问题是之后如何更新或调整函数的内部代码结构呢？

答案是无须更改。与其纠结于内部代码，不如简单定义一个封装原始函数的新函数。这样能简化参数值或结果的处理，并且无须理会原始函数的核心处理机制。

定义装饰器时会涉及高阶函数的两个阶段，如下所示。

- 在定义阶段，装饰器函数将封装基函数并返回封装后的函数。作为构建装饰器函数的一部分，在装饰过程中会处理一些一次性求值，例如计算比较复杂的默认值。
- 在求值阶段，封装函数会（并且通常可以）对基函数进行求值。封装函数可以对参数值进行预处理，也可以对返回值进行后处理（或者两者兼有）。使用封装函数也可以避免调用基函数，例如在管理缓存的情况下，封装可避免调用基函数产生的高昂开销。

简单的装饰器如下所示：

```
from functools import wraps
from typing import Callable, Optional, Any, TypeVar, cast

FuncType = Callable[..., Any]
F = TypeVar('F', bound=FuncType)

def nullable(function: F) -> F:
    @wraps(function)
    def null_wrapper(arg: Optional[Any]) -> Optional[Any]:
        return None if arg is None else function(arg)
    return cast(F, null_wrapper)
```

通常使用 `functools.wraps()`函数来确保被装饰的函数能保留原始函数的属性，例如复制 `__name__`和`__doc__`属性可以确保生成的装饰器函数具有原始函数的名称和文档字符串。

生成的复合函数，即装饰器定义中的 `null_wrapper()`函数，也是一种高阶函数。它以表达式的形式结合了原始函数，即可调用对象 `function()`，并保留了 `None` 值。在生成的 `null_wrapper()`函数中，原始的可调用对象 `function` 不再作为显式参数，而是能从 `null_wrapper()`函数定义的上下文中获取其值的自由变量。

装饰器函数的返回值是新创建的函数，名称与原始函数相同。装饰器只返回函数而不处理其中的数据，这一点很重要。装饰器使用的是元编程，即可以创建代码的代码。随后生成的 null_wrapper()函数可用于处理实际数据。

请注意，类型提示使用了 TypeVar 的特性来确保运用装饰器后得到的是 Callable 类型的对象。类型变量 F 的类型与原始函数绑定，而装饰器的类型提示声明了结果函数类型应与参数函数相同。通用的装饰器适用于各类函数，因此需要绑定一个类型变量。

可以运用@nullable 装饰器来创建复合函数，如下所示：

```
@nullable
def nlog(x: Optional[float]) -> Optional[float]:
    return math.log(x)
```

这会创建一个 nlog()函数，它是内置 math.log()函数的支持空值的版本。整个装饰过程返回了一个调用原始 nlog()函数的 null_wrapper()函数版本。结果命名为了 nlog()，并且具有封装后的函数和原始被封装函数的复合行为。

如下所示使用该复合 nlog()函数：

```
>>> some_data = [10, 100, None, 50, 60]
>>> scaled = map(nlog, some_data)
>>> list(scaled)
[2.302585092994046, 4.605170185988092, None, 3.912023005428146, 4.0943445622221]
```

我们对一个数据集应用了该函数。输入 None 值而输出 None，并且没有涉及任何异常处理。



这种示例并不适合单元测试。出于测试的目的，应对数值进行舍入。为此，还需创建一个支持空值的 round()函数。

使用装饰器表示法创建一个支持空值的舍入函数，如下所示：

```
@nullable
def nround4(x: Optional[float]) -> Optional[float]:
    return round(x, 4)
```

该函数使用和封装了 round()函数的部分功能来支持空值。

使用 Optional 类型定义，方便了模块 typing 描述支持空值的函数类型和支持空值的结果类型。定义 Optional[float]等同于 Union[None, float]，即 None 对象和 float 对象皆可使用。

还可以创建支持空值的舍入函数，如下所示：

```
nround4 = nullable(lambda x: round(x, 4))
```

请注意，我们没有在函数定义前使用装饰器，而是将装饰器用于一个匿名函数，这与在函数定义前添加装饰器的效果相同。

可以使用 round4()函数为 nlog()函数创建更好的测试用例，如下所示：

```
>>> some_data = [10, 100, None, 50, 60]
>>> scaled = map(nlog, some_data)
>>> [nround4(v) for v in scaled]
[2.3026, 4.6052, None, 3.912, 4.0943]
```

这里的结果与平台无关，因此便于文档测试。

对匿名函数使用类型提示可能有一定的挑战。以下代码说明了需要为此做些什么。

```
nround4l: Callable[[Optional[float]], Optional[float]] = (
    nullable(lambda x: round(x, 4))
)
```

变量 nround4l 的类型提示是一个带有[Optional[float]]参数列表的 Callable 对象，且其返回值类型是 Optional[float]。使用 Callable 类型提示仅适用于位置参数，对于存在关键字参数及其他复杂情况，请参阅 http://mypy.readthedocs.io/en/latest/kinds_of_types.html#extended-callable-types。

装饰器@nullable 假定所装饰的是一元函数（unary）。为了处理任意参数集合，需要重新考虑这个设计，创建一个更为通用的支持空值的装饰器。

第 14 章将介绍处理 None 值问题的一种替代方法。PyMonad 库定义了一个 Maybe 对象类，它既可以是某个合适的值，也可以是 None 值。

使用 functools 的 update_wrapper()函数

装饰器@wraps 利用 update_wrapper()函数来保留被封装函数的某些属性，通常这就完成了默认情况下我们需要的一切。该函数将原始函数的一组特定属性列表复制到了由装饰器创建的结果函数中。这组特定的属性列表是什么？它由一个全局模块定义。

函数 update_wrapper()依赖一个模块的全局变量来决定保留哪些属性。变量 WRAPPER_ASSIGNMENTS 定义了默认被复制的属性。这组默认被复制的属性列表是：

```
('__module__', '__name__', '__qualname__', '__doc__', '__annotations__')
```

很难对该列表做有意义的修改。def 语句的内部不允许进行简单的修改或变更，这一点需要注意。

如果要创建 callable 对象，可以用一个类来提供作为定义的一部分的一些附加属性。这可能导致装饰器必须将这些附加属性从原始被封装的 callable 对象复制到正在创建的封装函数中。然而，通过面向对象的类设计方法来进行这种修改，会比利用复杂装饰器技术简单一些。

11.2　横切关注点

装饰器背后的一条通用原则是可以从装饰器和使用该装饰器的原始函数中构建出复合函数。其思路是构建一个通用装饰器库，提供对常见关注点的实现。

我们通常把这些问题称为**横切**关注点，因为它们适用于多种函数。这些是我们希望通过装饰器实现的一次性设计，然后将它们应用于整个应用程序或框架中相关的类。

以装饰器函数定义为中心的关注点包括以下内容：

- 日志记录
- 审计
- 安全
- 处理不完整数据

例如，`logging` 装饰器可以将标准化消息写入应用程序的日志文件，审计装饰器可以写入数据库更新的相关详细信息，安全装饰器可以检查一些运行时上下文以确保登录用户拥有足够的权限。

对函数**支持空值**的封装便是横切关注点的一个例子。其中，我们希望一些函数处理 `None` 值后返回 `None` 值而不是引发异常。在某些数据不完整的应用中，我们可能希望以简单统一的方式处理行数据，而不必编写会分散注意力的大量 `if` 语句来处理缺失值。

11.3　复合设计

复合函数的常用数学表示法如下所示：

$$f \circ g(x) = f(g(x))$$

这里的思想是可以定义一个结合了另外两个函数 $f(y)$和 $g(x)$的新函数 $f \circ g(x)$ 。

Python 中复合函数的多行定义可以通过以下代码实现：

```
@f_deco
def g(x):
    something
```

生成的函数本质上等同于 $f \circ g(x)$。装饰器 `f_deco()`必须融合 `f()`的内部定义与给定的 `g()`来定义并返回复合函数。

实现细节显示了实际上 Python 提供的是一种略复杂的复合。封装器的结构有助于我们将 Python 装饰器的复合理解为以下形式：

$$\omega \circ g(x) = \omega_\beta \circ g \circ \omega_\alpha(x) = \omega_\beta(g(\omega_\alpha(x)))$$

应用于某个函数 $g(x)$的装饰器将包含一个封装函数 w。该封装器包括两部分，其中一部分 $\omega_\alpha(y)$用作被封装函数 $g(x)$的参数，而另一部分 $\omega_\beta(z)$作为被封装函数返回的结果。

下面给出具体示例，`something_wrapper()`装饰器的定义如下所示：

```
@wraps(argument_function)
def something_wrapper(*args, **kw):
    # The "before" part, w_α, applied to *args or **kw
    result = argument_function(*args, **kw)
    # The "after" part, w_β, applied to the result
    return after_result
```

这里显示了在原始函数的前面和后面可以注入额外处理的位置。这里强调了复合函数的抽象概念与 Python 的具体实现之间的一个重要区别：装饰器既可以创建 $f(g(x))$或 $g(f(x))$，也可以创建形式更复杂的 $f_\beta(g(f_\alpha(x)))$。装饰器语法无法完全描述所创建的复合函数的类型。

装饰器的真正价值在于封装函数中可以使用任何 Python 语句。装饰器可以使用 `if` 语句或 `for` 语句将函数转换为条件结构或循环结构。后面的示例会利用 `try` 语句对不良数据执行标准恢复操作。在这个通用框架中可以实现很多便利。

许多函数式编程都遵循 $f \circ g(x)=f(g(x))$设计模式。通过两个小函数来定义一个复合函数并不总是有用的。在某些情况下，分离这两个函数更有意义；而在其他情况下，我们可能想创建一个能汇总处理的复合函数。

创建 `map()`、`filter()`和 `reduce()`等常用的高阶函数的复合函数很容易。由于这些函数都很简单，因此复合函数通常很容易描述，并能提高程序的可读性。

例如，应用程序可能包含 `map(f, map(g, x))`函数，创建复合函数并使用 `map(f_g, x)`表达式来描述集合的复合可能更清楚。使用 `f_g = lambda x: f(g(x))`往往有助于将一个复杂应用解释为简单函数的复合。

值得注意的是，这两种技术在性能上相差不大。函数 `map()`是惰性的的，即对于两个 `map()`函数来说，从 `x` 中提取一项，会先经由 `g()`函数处理，然后由 `f()`函数处理。对于单个 `map()`函数来说，从 `x` 中提取一项后会交由复合函数 `f_g()`处理。

第 14 章将介绍这个问题的另一种解决方案，即通过单独的柯里化方法创建复合函数。

预处理不良数据

在一些 EDA 应用中，一个横切关注点是如何处理缺失的或无法解析的数值。货币值常混合有 `float`、`int` 和 `Decimal` 格式的数值，我们希望以统一的方式来处理它们。

在其他情况下，我们会碰到**不适用**或**不可用**的数据值，不应该让它们干扰主线程的计算。允许 Not Applicable 值在不引发异常的前提下传递给表达式，通常会比较方便。下面重点介绍三个针对不良数据的转换函数：bd_int()、bd_float()和 bd_decimal()。我们会在内置转换函数之前定义添加复合函数的特性。

简易的不良数据装饰器如下所示：

```
import decimal
from typing import Callable, Optional, Any, TypeVar, cast

FuncType = Callable[..., Any]
F = TypeVar('F', bound=FuncType)

def bad_data(function: F) -> F:
    @wraps(function)
    def wrap_bad_data(text: str, *args: Any, **kw: Any) -> Any:
        try:
            return function(text, *args, **kw)
        except (ValueError, decimal.InvalidOperation):
            cleaned = text.replace(",", "")
            return function(cleaned, *args, **kw)
    return cast(F, wrap_bad_data)
```

该函数封装了一个转换函数（function），以便在第一次转换碰到不良数据时尝试第二次转换。第二次转换将在删除,字符后进行。此封装器将*args 和**kw 参数传递给被封装的函数，使得被封装的函数可以获得额外的参数。

将类型变量 F 与 function 参数的原始定义绑定。定义装饰器返回相同类型（F）的函数。使用函数 cast()向 mypy 工具给出提示——封装器不会更改被封装函数的签名。

可以使用此封装器创建能感知不良数据的函数，如下所示：

```
bd_int = bad_data(int)
bd_float = bad_data(float)
bd_decimal = bad_data(Decimal)
```

这样就可以创建出一组函数，它们既可以转换良好数据，也可以针对特定类型的不良数据进行有限的数据清理。

对于某些可调用对象，很难写出它们的类型提示。尤其是 int()函数具有可选关键字参数，所以其类型提示会比较复杂。有关为可调用对象创建复杂类型签名的指南，请参阅 http://mypy.readthedocs.io/en/latest/kinds_of_types.html?highlight=keyword#extended-callable-types。

使用 bd_int()函数的示例如下：

```
>>> bd_int("13")
13
>>> bd_int("1,371")
```

```
1371
>>> bd_int("1,371", base=16)
4977
```

上面运用了 bd_int()函数对字符串进行了简易转换，它也能接收含特定标点符号的字符串。此外每个转换函数可接收额外的参数。

有时需要一个更为灵活的装饰器，或者添加一些特性，例如处理各种数据清洗方案的功能。简单删除并不总是理想的。有时还要删除$或°符号。稍后将介绍更复杂的参数化装饰器。

11.4 向装饰器添加参数

一个常见的需求是使用额外的参数去自定义一个装饰器。我们可以做一些更复杂的处理，而不只是简单地创建一个复合函数 $f \circ g(x)$。使用参数化装饰器可以创建$(f(c) \circ g)(x)$。前面已经使用了参数 c 作为创建封装器 $f(c)$的一部分。这个参数化的复合函数 $f(c) \circ g$ 之后便可以和实际数据 x 一同使用了。

在 Python 中，可以编写如下代码：

```
@deco(arg)
def func(x):
    something
```

这里包含两个步骤。首先是将参数应用于抽象装饰器以创建具体的装饰器，然后将该具体的装饰器，即参数化的函数 deco(arg)，应用于基函数定义中来创建装饰器函数。

该装饰器函数的效果如下所示：

```
def func(x):
    return something(x)
concrete_deco = deco(arg)
func= concrete_deco(func)
```

实际上做了以下三件事：

(1) 定义一个函数 func()；

(2) 对参数 arg 应用抽象装饰器 deco()来创建一个具体装饰器 concrete_deco()；

(3) 对基函数应用该具体装饰器 concrete_deco()来创建函数的装饰器版本，实际上就是 deco(arg)(func)。

带参数的装饰器包含了对最终函数的间接构造。这似乎已经超越了单纯的高阶函数，进入了更为抽象的领域，即创建高阶函数的高阶函数。

可以扩展能感知不良数据的装饰器来创建一个更灵活的转换器。下面定义一个@bad_char_remove 装饰器，它以待删除字符为参数。这个参数化的装饰器如下：

11

```
import decimal
def bad_char_remove(
        *char_list: str
    ) -> Callable[[F], F]:
    def cr_decorator(function: F) -> F:
        @wraps(function)
        def wrap_char_remove(text: str, *args, **kw):
            try:
                return function(text, *args, **kw)
            except (ValueError, decimal.InvalidOperation):
                cleaned = clean_list(text, char_list)
                return function(cleaned, *args, **kw)
        return cast(F, wrap_char_remove)
    return cr_decorator
```

一个参数化的装饰器具有两个内部函数定义。

- 抽象装饰器:cr_decorator 函数会将其绑定的自由变量 char_list 变成具体的装饰器。随后返回该装饰器将其应用于函数，这将返回一个封装在 wrap_char_remove 函数内的函数。这里作为类型提示的类型变量 F 声明了封装操作将保留被封装函数的类型。
- 被装饰的封装器：wrap_char_remove 函数会用封装版本替代原始函数。由于使用了 @wraps 装饰器，因此被封装的基函数的名字将替代新函数的__name__属性（以及其他属性）。

整个装饰器 bad_char_remove() 函数的任务是将参数绑定到抽象装饰器并返回具体装饰器。类型提示阐明了返回值是一个能将 Callable 函数转换为另一个 Callable 函数的 Callable 对象。之后根据语言规则，会将具体装饰器应用于之后的函数定义。

下面的 clean_list() 函数用于删除在给定列表中的所有字符。

```
from typing import Tuple
def clean_list(text: str, char_list: Tuple[str, ...]) -> str:
    if char_list:
        return clean_list(
            text.replace(char_list[0], ""), char_list[1:])
    return text
```

其中的规则十分简单，因此使用了递归方法，也可以把它优化成一个循环。

可以使用@bad_char_remove 装饰器创建转换函数，如下所示·：

```
@bad_char_remove("$", ",")
def currency(text: str, **kw) -> Decimal:
    return Decimal(text, **kw)
```

上面使用了装饰器来封装 currency() 函数。函数 currency() 的本质特征是对 decimal.Decimal 构造器的引用。

这个 currency() 函数可以处理不同的数据格式了。

```
>>> currency("13")
Decimal('13')
>>> currency("$3.14")
Decimal('3.14')
>>> currency("$1,701.00")
Decimal('1701.00')
```

接下来可以使用一个相对简单的 `map(currency, row)`方法处理输入数据，以便将源数据从字符串转换为可用的 `Decimal` 值了。错误处理语句 `try:/except:`已经隔离到了一个用于构建复合转换函数的函数中。

可以使用类似的设计来创建可以容错空值的函数。这些函数将使用类似的 `try:/except:` 封装器，但只简单地返回 `None` 值。

11.5 实现更复杂的装饰器

在 Python 中，可以使用如下命令创建更复杂的函数：

```
@f_wrap
@g_wrap
def h(x):
    something
```

在 Python 中，可以任意堆叠装饰器来修改其他装饰器返回的结果，其含义有点类似于 $f \circ g \circ h(x)$。最终得到的函数名称仍然是 $h(x)$。由于存在这种潜在的混淆，因此在创建包含深度嵌套装饰器的函数时要格外小心。如果仅为处理一些横切关注点，那么应该把每个装饰器都设计成能处理单独问题，而且不会造成太多混乱。

另外，如果使用装饰器来创建复合函数，那么最好使用以下几类定义：

```
from typing import Callable

m1: Callable[[float], float] = lambda x: x-1
p2: Callable[[float], float] = lambda y: 2**y
mersenne: Callable[[float], float] = lambda x: m1(p2(x))
```

这里每个变量都有一个类型提示用于定义关联函数。`m1`、`p2` 和 `mersenne` 这 3 个函数都具有 `Callable[[float], float]`类型提示，表明该函数将接收一个可以被强制转换为浮点数并返回数字的数字。使用类似于 `F_float = Callable[[float], float]`的类型定义可以避免类型定义上过分简单的重复。

如果函数规模比简单表达式大，强烈建议使用 `def` 语句，这里使用匿名函数对象是比较罕见的情况。

尽管装饰器可以实现很多事情，但需要弄清楚使用装饰器是否能编写出清晰、简洁且易读的

代码。在处理横切关注点时，装饰器特性通常与被装饰函数存在本质区别，它极大地简化了被封装的函数。通过装饰器添加日志记录、调试或安全检查是常见的做法。

然而对于过于复杂的装饰器设计，一个严重的后果是难以提供适当的类型提示。当类型提示退化为只能使用 `Callable` 和 `Any` 时，这样的设计可能会变得晦涩难懂。

11.6　复杂设计注意事项

在数据清洗的案例中，简单地删除杂散字符可能还不够。当处理地理位置数据时，我们会碰到各种输入格式，包括简单的度数（`37.549016197`）、度数和分数（`37° 32.94097′`）以及度–分–秒（`37° 32′ 56.46″`）。当然，可能还有更微妙的清洗问题，比如一些设备会用 Unicode 编码为 U+00BA 的字符 º 而不是编码为 U+00B0 且与之形似的度数字符°作为输出。

因此，通常需要提供与转换函数绑定的单独清洗函数。该函数会对在格式上与经纬度格式相差较大的输入数据进行更复杂的转换处理。

那么该如何实现呢？对此有多种选择，例如简单的高阶函数，然而装饰器的效果却不太好。下面展示一个基于装饰器的设计，来说明装饰器的局限性。

这里要求有两个正交设计的考量，如下所示：

- 输出转换（`int`、`float` 和 `Decimal`）；
- 输入清洗（清除杂散字符，重新格式化坐标）。

理想情况下，其中一个可以是被封装的基本函数，另一个则以封装器的形式被引入。很难说清楚该选择哪个作为基本函数，哪个作为封装器，原因之一是上述示例比单一的两层复合要复杂一些。

考虑到上述示例，似乎应该将其看作一个三层复合函数：

- 输出转换（`int`、`float` 和 `Decimal`）；
- 输入清洗——简单替换或更复杂的多字符替换；
- 函数首先尝试转换，并执行清洗来应对异常，然后再次尝试转换。

这里执行尝试转换与重试的第三部分，才是真正意义上的封装器，同时也是复合函数的一部分。正如之前提到的，一个封装器包含了一个参数阶段和一个可以返回值的值，分别称为 $\omega_\alpha(x)$ 和 $\omega_\beta(x)$。

下面用此封装器基于两个额外函数创建一个复合函数。设计上有两种选择，一种是将清洗函数作为转换装饰器的参数，如下所示：

```
@cleanse_before(cleanser)
def conversion(text):
    something
```

这种设计声明了转换函数是其核心，而清洗函数作为辅助细节实现，它会修改程序行为，但会保留转换函数的原始意图。

也可以将转换函数作为清洗函数装饰器的一个参数，如下所示：

```
@then_convert(converter)
def cleanse(text):
    something
```

第二种设计表明了清洗函数是其核心，而转换函数作为辅助细节实现。这有点令人困惑，因为清洗函数的类型通常是 `Callable[[str], str]`，而转换函数的类型 `Callable[[str], some other type]`是整个被封装函数所需的类型。

尽管这两种方法都可以创建出可用的复合函数，但第一种方法有个重要的优势，即 `conversion()`函数的类型签名同时也是生成的复合函数的类型签名。这凸显了装饰器的一个通用设计模式，即被装饰的函数其类型最容易保留。



当面临定义复合函数的不同选择时，保留被装饰函数的类型提示是很重要的一点。

因此，我们首选`@cleanse_before(cleaner)`风格的装饰器。该装饰器形式如下：

```
def cleanse_before(
        cleanse_function: Callable
    ) -> Callable[[F], F]:
    def abstract_decorator(converter: F) -> F:
        @wraps(converter)
        def cc_wrapper(text: str, *args, **kw) -> Any:
            try:
                return converter(text, *args, **kw)
            except (ValueError, decimal.InvalidOperation):
                cleaned = cleanse_function(text)
                return converter(cleaned, *args, **kw)
        return cast(F, cc_wrapper)
    return abstract_decorator
```

上面定义了一个三层装饰器，其核心是使用了 `converter()`函数的 `cc_wrapper()`函数。如果该操作失败，那么它会使用给定的 `cleanse_function()`函数并再次尝试使用 `converter()`函数。具体的装饰器函数 `abstract_decorator()`在 `cleanse_function()`函数和 `converter()`函数之外构建了一个 `cc_wrapper()`函数。该具体装饰器以 `cleanse_function()`函数作为自由变量，并由装饰器接口 `cleanse_before()`创建，后者由 `cleanse_function()`函数定制。

其中的类型提示强调了`@cleanse_before` 装饰器的作用。它会接收某个 `Callable` 函数，

即这里的 cleanse_function，并创建一个函数 Callable[[F], F]，该函数会把一个函数转换为一个被封装的函数。此过程有助于我们理解参数化装饰器的工作机制。

接下来构建一个稍灵活的清洗和转换函数 to_int()，如下所示：

```
def drop_punct2(text: str) -> str:
    return text.replace(",", "").replace("$", "")

@cleanse_before(drop_punct)
def to_int(text: str, base: int = 10) -> int:
    return int(text, base)
```

清洗函数装饰了整数转换函数。本例中，清洗函数删除了$和,字符。该清理函数封装了整数转换函数。

上面定义的 to_int()函数利用了内置的 int()函数。为了避免使用 def 语句，可如下定义：

```
to_int2 = cleanse_before(drop_punct)(int)
```

这里使用了 drop_punct()函数封装内置的 int()转换函数。借助 mypy 工具中的 reveal_type()函数，可以看到 to_int()函数的类型签名与内置 int()函数的类型签名是相匹配的。

如下所示使用改进后的整数转换函数：

```
>>> to_int("1,701")
1701
>>> to_int("97")
97
```

对于被装饰的函数 to_int()来说，底层 int()函数的类型提示已经重写（和简化）了。这就是试图使用装饰器封装内置函数的一个结果。

复杂的参数化装饰器让设计如履薄冰。装饰器模型似乎不太适合这种设计，而复合函数的定义似乎比构建装饰器所需的机制更清晰。

通常，当我们想在给定函数（或者类）中加入一些相对简单和固定的内容时，装饰器很适用。当这些额外的内容对于应用程序代码的含义不重要，而被视作基础结构或作为支撑时，装饰器也是很重要的。

当涉及多个正交设计方面时，我们可能希望得到一个可调用的类定义，它具有各种插件的策略对象。这个类的定义可能比功能相同的装饰器更简单。替代装饰器的另一个方法是创建高阶函数。在某些情况下，具有各种参数组合的偏函数会比装饰器更简单。

横切关注点的典型示例包括日志记录和安全测试。可以将这些特性看作不特定于某个问题领域的后台处理。当我们能自如地使用装饰器时，它才能算是好用的设计技术。

11.7 小结

本章介绍了两种装饰器：不带参数的装饰器和参数化的装饰器，还介绍了装饰器是如何参与函数之间的间接复合的，即装饰器用一个函数（定义在装饰器内部）封装另一个函数。

使用 `functools.wraps()`装饰器可以确保装饰器能正确复制被封装函数的属性，这一点应该作为编写装饰器的一个考量。

下一章将介绍多进程技术和多线程技术，这些包在函数式编程环境中特别有用。当我们消除复杂的共享状态并围绕非严格运算展开设计时，便可以利用并行性来提高性能了。

第 12 章

multiprocessing 和 threading 模块

当消除了复杂的共享状态并围绕非严格执行处理展开设计时，便可以利用并行性来提高性能了。本章将介绍多进程技术和多线程技术。对于允许惰性求值的算法，Python 的库包会特别有用。

其中心思想是在一个或多个进程内的多个线程之间分配一个函数式程序。如果已经有了合理的函数式设计，那么就可以避免应用程序组件之间的复杂交互，因为可以通过函数接收参数值并生成结果。这是进程或线程的一个理想结构。

本章将讨论以下几个主题。

- 函数式编程和并发的总体思想。
- 当考虑计算核心、CPU 和操作系统级的并行时，并发实际意味着什么？需要注意的是，并发并不会神奇地加速糟糕的算法。
- 使用内置的 `multiprocessing` 模块和 `concurrent.futures` 模块。这些模块支持多种并行执行技术。`dask` 包也可以完成很多类似的工作。

本章将重点讨论进程级的并行性而非线程级的。利用进程级的并行性让我们可以完全忽略 Python 的**全局解释器锁**（global interpreter lock，GIL）。有关 Python GIL 的更多信息，请参阅 https://docs.python.org/3.3/c-api/init.html#thread-state-and-the-global-interpreter-lock。

12.1 函数式编程和并发

当执行任务之间没有任何依赖关系时并发处理最为高效。并发（或**并行**）编程开发的最大难点在于协调对共享资源的更新。

在遵循函数式设计模式的前提下，我们往往避免设计状态化的程序。函数式设计应当尽量减少或消除对共享对象的并发更新。如果可以设计出以惰性求值和非严格求值为核心的软件，那么

同样可以设计出并发求值的软件。这样便能实现**高度并行**的设计[①]，其中大多数工作都可以并发完成，而计算之间的交互则很少或根本没有。

编程的核心是运算之间的相互依赖。在表达式`2*(3+a)`中，必须先计算子表达式`(3+a)`。表达式的最终值取决于两个运算的先后顺序。

在处理集合对象时，经常会遇到这样的情况：集合中各项的处理顺序无关紧要。考虑以下两个例子：

```
x = list(func(item) for item in y)
x = list(reversed([func(item) for item in y[::-1]]))
```

即使这两个命令对表达式`func(item)`的求值顺序是相反的，结果仍是相同的。这只有在每一次`func(item)`的求值都是独立的且没有副作用的情况下才可行。

下面的命令片段效果也相同：

```
import random
indices = list(range(len(y)))
random.shuffle(indices)
x = [None]*len(y)
for k in indices:
    x[k] = func(y[k])
```

上述示例中的求值顺序是随机的。由于对`func(y[k])`的每一次求值都是相对独立的，因此求值顺序并不重要。许多算法都允许这种非严格求值。

12.2 并发的意义

在只有单个处理器和一个计算核心的小型计算机中，处理器这个唯一的核心将所有计算串行化了。整个操作系统会通过巧妙的时间片分配交错出多个进程和多个线程的效果。

在多 CPU 或单 CPU 多计算核心的计算机上，可以对 CPU 指令进行一些实际的并发处理。其他所有类型的并发都是在操作系统层通过时间片模拟出来的。一台 macOS X 笔记本电脑可以有 200 个并发进程共享 CPU，这里的进程数比可用的计算核心个数多得多。可见总体而言操作系统通过时间片负责绝大部分比较显见的系统并发行为。

12.2.1 边界条件

设想有一个算法，其算法复杂度为 $O(n^2)$。假设它有一个包含 1000 字节 Python 代码的内循环，那么在处理 10 000 个这样的对象时，需要执行 1000 亿次 Python 运算。我们称其为基本处

① 也称“天然并行”。——译者注

理预算。只要我们觉得会有所帮助，那么就可以分配尽可能多的进程和线程，但处理预算并不会改变。

单个 CPython 字节码的执行时间不易统计。然而，在 macOS X 笔记本电脑上，长期平均值显示每秒可以执行 60MB 左右的代码。这意味着执行 1000 亿的字节码大约需要 1666 秒或 28 分钟。

如果有一台双处理器四核计算机，那么有可能将运行时间缩减到原来总时间的 25%，即大约 7 分钟。这里我们假定将任务划分给了 4 个（或更多）独立的操作系统进程。

其中一个很重要的考量是 1000 亿字节码的总预算是无法改变的。并行性并不会神奇地减少工作负载，它只能通过改变对任务的调度来减少耗时。

如果切换到一个更好的、复杂度为 $O(n \log n)$的算法上，那么其工作负载可以从 1000 亿次运算减少到 1.33 亿次运算，运行时间大约是 2 秒。如果平均分配到 4 个核上，甚至在 516 毫秒内就能得到结果。并行性无法像改变算法这样显著提升性能。

12.2.2　进程或线程间共享资源

操作系统确保了进程之间很少或根本没有交互。在创建进程间必须交互的多进程应用时，必须显式地共享一个公共的操作系统资源。这可以是一个公共文件、一个共享内存的特定对象，或是进程间具有共享状态的一个信号量。进程本质上是彼此独立的，它们之间的交互只是一种特殊情况。

相反，多线程是单个进程的一部分，且一个进程中的所有线程共享操作系统资源。我们可以破例获取一些线程本地内存，这些内存可以在不干扰其他线程的情况下自由地写入。在线程本地内存之外，写内存操作会以不可预知的顺序设置进程的内部状态，因此必须通过显式加锁来避免。如前所述，指令执行的整个顺序严格来讲很少是并发的。并发线程和并发进程的指令通常会以不可预知的顺序在计算核心之间交错。线程化意味着可以对共享变量进行破坏性更新，并且需要通过加锁才能进行互斥访问。

裸机硬件层存在一些更复杂的写内存情况。关于写内存问题的更多信息，可参考维基百科词条 memory disation。

在设计多线程应用程序时，并发对象更新的存在可能会造成混乱。加锁是避免并发写共享对象的一种方法。通常避免使用共享对象也是一种可行的设计技术。第二种技术——避免写共享对象更适用于函数式编程。

在 CPython 中，GIL 用于确保操作系统的线程调度不会干扰 Python 内部数据结构的维护。实际上，GIL 将调度的粒度从机器指令变成了 Python 虚拟机运算组。

GIL 在确保数据结构完整性方面所产生的影响通常可以忽略不计，选用的算法对性能影响最大。

12.2.3　从何处受益

并发处理对执行大量计算和相对较少 I/O 操作的程序帮助不大。如果一个计算预计需要 28 分钟完成，那么以不同方式交错运算并不会有显著的影响。使用 8 个计算核心也许只能缩减大约 1/8 的时间。实际节省的时间取决于操作系统和语言本身的开销，而它们是很难预估的。引入并发对性能的影响不及使用更好的算法。

当计算涉及大量 I/O 时，通过交错 CPU 运算和 I/O 请求能**显著**提升性能。其核心思想是在等待操作系统完成其他数据 I/O 的同时计算一部分数据。由于 I/O 通常需要等待很长时间，因此一个八核处理器可以交错执行大量并发的 I/O 请求。

交叉计算和 I/O 的方法有两种，如下所示。

- 可以创建一个阶段化处理的管道。每项数据必须经由读取、过滤、计算、聚合和写入等所有阶段。多并发阶段是指每个阶段处理不同的数据对象。阶段之间通过时间分片来实现计算和 I/O 的交错。
- 可以创建一个并发工作池，每个 worker 负责对一个数据项执行所有计算。数据项分配给池中所有 worker，由 worker 生成结果。

这些方法之间的区别不是特别明显，中间存在不是非此即彼的模糊区域。通常创建一个混合的结合体，用工作池来加速管道中的某个阶段，使其和管道中的其他阶段执行速度一样快。这里可以参照一些形式体系在一定程度上简化并发程序的设计。**通信顺序进程**（communicating sequential processes，CSP）范式可以辅助设计消息传递的应用程序。可以使用类似于 `pycsp` 这样的包向 Python 中添加 CSP 形式体系。

I/O 密集型程序通常可以从并发处理中获得最大好处，其思想是交错 I/O 和计算。CPU 密集型程序从并行处理中获得的好处相对较小。

12.3　使用多进程池和任务

并发是一种**非严格**求值形式，即无法预知操作的实际顺序。`multiprocessing` 包引入了 `Pool` 对象的概念。一个 `Pool` 对象包含许多工作进程，且这些进程能并发执行。这个包允许操作系统调度和时间分片交错地执行多个进程，旨在使整个系统尽可能地满负荷运作。

为了充分利用这种功能，需要将应用程序分解为支持非严格执行的组件。整个应用程序必须由能以非确定顺序处理的离散任务来构建。

例如，一个通过网页抓取从互联网收集数据的应用通常可以通过并行处理实现优化。我们可以创建一个具有多个同等 worker 的 `Pool` 对象来进行网站抓取。URL 形式的任务分配给每个 worker 供其分析。

分析多个日志文件的应用也能实现并行化。可以创建一个包含分析 worker 的 `Pool` 对象，将每个日志文件分配给一个分析 worker，这就使得 `Pool` 对象中各个 worker 能并行地进行读取和分析。每个 worker 都会执行 I/O 和计算，但一个 worker 可以在其他 worker 等待 I/O 完成的同时执行分析。

由于性能获益取决于时间难以预测的输入和输出操作，因此往往需要对多进程进行大量试验。改变工作池的大小和测量运行时间是设计并发应用程序的重要部分。

12.3.1　处理大量大型文件

下面是一个多进程应用程序的示例。我们将从 Web 日志文件中提取**通用日志格式**（common log format，CLF），这是 Web 服务器的访问日志较常用的一种格式。文件中的行往往很长，但因书的页边距换行后会如下所示：

```
99.49.32.197 - - [01/Jun/2012:22:17:54 -0400] "GET /favicon.ico
 HTTP/1.1" 200 894 "-" "Mozilla/5.0 (Windows NT 6.0)
 AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52
 Safari/536.5"
```

通常需要分析大量文件。许多独立文件的存在意味着并发有助于信息抓取。

分析过程可分解为两个比较宽泛的功能。处理的第一阶段是对日志文件进行必要的解析，以收集相关信息片段。我们把该解析过程细分为 4 个阶段，如下所示：

(1) 读取多个源日志文件的所有行数据；
(2) 然后为文件集合中每一行日志条目创建简单的命名元组对象；
(3) 解析日期和 URL 等复杂字段的详细信息；
(4) 从日志中排除不感兴趣的路径名，也可以将其看作只解析感兴趣的路径名。

解析阶段过后便可以执行大量分析操作了。下面通过一个用于计算特定路径出现频次的简单分析来演示 `multiprocessing` 模块。

第一部分是读取包含大部分输入处理的源文件。Python 利用文件迭代器把对缓冲数据的请求转化为了底层操作系统的请求。操作系统的每个请求都意味着进程必须等待直到数据可用。

显然，如果同时交错其他操作，它们就无须等待 I/O 完成了。交错执行的数据范围可以从独立的行数据到整个文件数据。首先考虑交错执行整个文件，因为这相对容易实现。

解析 Apache CLF 文件的函数式设计如下所示：

```
data = path_filter(
    access_detail_iter(
        access_iter(
            local_gzip(filename))))
```

较大的解析问题分解为了许多函数，由这些函数负责处理解析问题的每个部分。函数 `lcoal_gzip()`从本地缓存的 GZIP 文件中读取行数据。函数 `access_iter()`在访问日志中为每一行创建了一个 `NamedTuple` 对象。函数 `access_detail_iter()`扩展到一些更难解析的字段。最后，函数 `path_filter()`将丢弃一些分析价值不大的路径和文件扩展名。

如下所示的管道处理有助于将这种设计可视化：

```
(local_gzip(filename) | access_iter | access_detail_iter | path_filter) > data
```

这里使用了 shell 的管道符号(|)将数据从一个进程传递至另一个进程。Python 内置的 `pipes` 模块支持构建实际的 shell 管道来利用操作系统的多进程能力。其他包（如 `pipetools` 或 `pipe`）提供了类似的可视化复合函数的方法。

12.3.2 解析日志文件之收集行数据

解析大量文件的第一步是读取每个文件并生成一个简单的行序列。由于日志文件是以`.gzip` 格式保存的，因此需要使用 `gzip.open()`函数打开每个文件，而不是使用 `io.open()`函数或 `__builtins__.open()`函数。

如下面的命令片段所示，函数 `local_gzip()`会从本地缓存的文件中读取行数据。

```
from typing import Iterator
def local_gzip(pattern: str) -> Iterator[Iterator[str]]:
    zip_logs = glob.glob(pattern)
    for zip_file in zip_logs:
        with gzip.open(zip_file, "rb") as log:
            yield (
                line.decode('us-ascii').rstrip()
                for line in log)
```

上述函数遍历了与给定模式匹配的所有文件。对于每个文件，生成的值是一个生成器函数，它会遍历该文件中的所有行数据。前面已经封装了一些东西，其中包括文件通配符匹配，打开以`.gzip` 格式压缩的日志文件的实现细节，以及将一个文件分解为不包含任何尾部换行符（`\n`）的行序列。

这里的基本设计模式是生成的值是每个文件的生成器表达式。可以把上述函数重写为一个函数和一个映射，该映射将这个特定函数应用于每个文件。对于需要识别个别文件的罕见情况，这种方法非常有用。在某些情况下，可以使用 `yield from` 对其进行优化，这会使得所有日志文件形似单个行数据流。

还有其他几种方法可以得到类似的输出，例如下面给出了上述示例中内层 `for` 循环的一个替代版本，其中的 `line_iter()`函数也能得到给定文件的行数据。

```
def line_iter(zip_file: str) -> Iterator[str]:
    log = gzip.open(zip_file, "rb")
    return (line.decode('us-ascii').rstrip() for line in log)
```

函数 line_iter()使用了 gzip.open()函数和一些行清理操作。可以通过映射将 line_iter()函数应用于模式匹配的所有文件，如下所示：

```
map(line_iter, glob.glob(pattern))
```

尽管这个替代版本很简洁，但其缺点是打开的文件对象必须一直等待，直到没有更多文件引用才能被当作垃圾进行回收。在处理大量文件时，这会产生不必要的开销。出于这个原因，我们将考虑使用之前讲的 local_gzip()函数。

上述映射的替代方案有一个明显优势：它和 multiprocessing 模块相得益彰。我们可以创建一个工作池并将任务（如文件读取）映射至该进程池中。这样做让我们可以并行读取这些文件，而打开的文件对象也将作为独立进程的一部分。

这个设计的扩展包含了第二个函数，它使用 FTP 传输来自 Web 主机的文件。当从 Web 服务器收集到文件后，便可以使用 local_gzip()函数来分析它们了。

函数 access_iter()会使用 local_gzip()函数的结果来为描述文件访问的源文件中的每一行创建命名元组。

12.3.3　解析日志行为命名元组

一旦访问了每个日志文件的所有行数据，就可以提取所描述的详细访问信息了。下面使用正则表达式来分解每一行，以此构建出 namedtuple 对象。

使用如下正则表达式解析 CLF 文件的每一行。

```
import re
format_pat = re.compile(
    r"(?P<host>[\d\.]+)\s+"
    r"(?P<identity>\S+)\s+"
    r"(?P<user>\S+)\s+"
    r"\[(?P<time>.+?)\]\s+"
    r'"(?P<request>.+?)"\s+'
    r"(?P<status>\d+)\s+"
    r"(?P<bytes>\S+)\s+"
    r'"(?P<referer>.*?)"\s+' # [SIC]
    r'"(?P<user_agent>.+?)"\s*'
)
```

可以使用这个正则表达式将每一行分解为包含 9 个独立数据元素的字典。使用[]和"来分隔以 time、request、referrer 以及 user_agent 等为参数的复杂字段，可以轻松地将文本转换为 NamedTuple 对象。

可以将每个独立访问概括为 NamedTuple 的子类，如下所示：

```
from typing import NamedTuple
class Access(NamedTuple):
    host: str
    identity: str
    user: str
    time: str
    request: str
    status: str
    bytes: str
    referer: str
    user_agent: str
```



上面竭力确保了 NamedTuple 的字段名与记录中每一部分的(?P<name>)构造体中的正则表达式组名称相匹配，以便于将解析后的字典转换为元组供后续处理。

下面的 access_iter()函数要求每个文件都表示为基于文件行的迭代器：

```
from typing import Iterator
def access_iter(
        source_iter: Iterator[Iterator[str]]
    ) -> Iterator[Access]:
    for log in source_iter:
        for line in log:
            match = format_pat.match(line)
            if match:
                yield Access(**match.groupdict())
```

函数 local_gzip()的输出是一组序列的序列。外层序列基于各个日志文件。对于每个文件，都有一个嵌套的可迭代行序列。如果某一行匹配给定的模式，那它就是某种文件访问。可以通过 match 字典创建 Access 命名元组。不匹配的行数据会被静默丢弃。

基本的设计模式是根据解析函数的结果构建不可变对象。在本例中，解析函数是一个正则表达式匹配器。其他类型的解析也适用于这种设计模式。

还有一些替代方法可以做到这一点。例如可以使用 map()函数，如下所示：

```
def access_builder(line: str) -> Optional[Access]:
    match = format_pat.match(line)
    if match:
        return Access(**match.groupdict())
    return None
```

上例中的替代函数只包含必要的解析和 Access 对象的构造。它会返回一个 Access 对象或者 None 对象。下面使用该函数将日志文件扁平化为单个 Access 对象流。

```
filter(
    None,
    map(
```

```
            access_builder,
            (line for log in source_iter for line in log)
        )
    )
```

这展示了将 local_gzip()函数的输出转换为一系列 Access 实例的过程。在本例中，我们将 access_builder()函数应用于具有可迭代结构的嵌套迭代器，该结构是通过读取文件集合得到的。函数 filter()从 map()函数的结果中移除了 None 对象。

这里重点展示了许多可用于解析文件的函数式方法。第 4 章介绍了非常简单的解析方法，而这里运用各种技术实现了更为复杂的解析。

12.3.4 解析 Access 对象的其他字段

对于构成访问日志文件行数据的 9 个字段，此前创建的 Access 对象不会对其内部的一些元素进行分解。我们将通过整体分解将这些数据项分别解析为高层字段。单独执行这些解析操作可以简化每个处理阶段。这也让我们无须破坏日志分析的通用结构而替换整个处理过程中的一小部分。

下一阶段解析得到的对象 AccessDetails 是 NamedTuple 的一个子类，并且它封装了原始的 Access 元组。该对象使用如下一些额外的字段来分别表示解析细节：

```
from typing import NamedTuple, Optional
import datetime
import urllib.parse

class AccessDetails(NamedTuple):
    access: Access
    time: datetime.datetime
    method: str
    url: urllib.parse.ParseResult
    protocol: str
    referrer: urllib.parse.ParseResult
    agent: Optional[AgentDetails]
```

属性 access 是原始的 Access 对象，即简单字符串的集合。属性 time 是解析后的 access.time 字符串。属性 method、url 和 protocol 通过分解 access.request 字段得到。属性 referrer 是一个解析后的 URL。

属性 agent 还可以分解为更细粒度的字段。非常规浏览器或网站爬虫会生成一个无法解析的 agent 字符串，因此该属性的类型提示为 Optional。

以下属性构成了 NamedTuple 类的子类 AgentDetails。

```
class AgentDetails(NamedTuple):
    product: str
```

```
    system: str
    platform_details_extensions: str
```

这些字段反映了描述代理最常用的语法。尽管这方面存在相当大的差异，但此特定子集似乎相当普遍。

以下 3 个解析器用于精细分解字段：

```
from typing import Tuple, Optional
import datetime
import re

def parse_request(request: str) -> Tuple[str, str, str]:
    words = request.split()
    return words[0], ' '.join(words[1:-1]), words[-1]

def parse_time(ts: str) -> datetime.datetime:
    return datetime.datetime.strptime(
        ts, "%d/%b/%Y:%H:%M:%S %z"
    )

agent_pat = re.compile(
    r"(?P<product>\S*?)\s+"
    r"\((?P<system>.*?)\)\s*"
    r"(?P<platform_details_extensions>.*)"
)

def parse_agent(user_agent: str) -> Optional[AgentDetails]:
    agent_match = agent_pat.match(user_agent)
    if agent_match:
        return AgentDetails(**agent_match.groupdict())
    return None
```

我们为 HTTP 请求、时间戳和用户代理信息编写了 3 个解析器。日志中的请求值通常是由 3 个词构成的字符串，如 `GET /some/path HTTP/1.1`。函数 `parse_request()`提取了这 3 个以空格分隔的值。如果路径中也包含空格，那么提取第一个词和最后一个词分别作为方法和协议，剩下的词则作为路径的一部分。

时间解析委派给了 `datetime` 模块，并且 `parse_time()`函数中提供了适当的格式。

解析用户代理比较有挑战性。对此有许多方法，我们为 `parse_agent()`函数选择了一种常用的处理方法。如果用户代理文本与给定的正则表达式匹配，那么使用 `AgentDetails` 类的属性。如果用户代理信息与正则表达式不匹配，那么使用 `None` 值代替。原始文本可以从 `Access` 对象中获取。

基于给定的 `Access` 对象，我们将使用这 3 个解析器构建 `AccessDetails` 实例。函数 `access_detail_iter()`的主体如下所示：

```
from typing import Iterable, Iterator
def access_detail_iter(access_iter: Iterable[Access]) -> Iterator[AccessDetails]:
    for access in access_iter:
        try:
            meth, url, protocol = parse_request(access.request)
            yield AccessDetails(
                access=access,
                time=parse_time(access.time),
                method=meth,
                url=urllib.parse.urlparse(url),
                protocol=protocol,
                referrer=urllib.parse.urlparse(access.referer),
                agent=parse_agent(access.user_agent)
            )
        except ValueError as e:
            print(e, repr(access))
```

我们使用了与先前 access_iter()函数类似的设计模式。解析某个输入对象，基于结果构建了一个新对象。新的 AccessDetails 对象会封装之前的 Access 对象。利用这种技术让我们不仅可以使用不可变对象，还能包含更多详细信息。

这个函数本质上是从一个 Access 对象到一个 AccessDetails 对象的映射。另一种替代设计如下，它使用了相对高阶的 map()函数。

```
from typing import Iterable, Iterator
def access_detail_iter2(
        access_iter: Iterable[Access]
    ) -> Iterator[AccessDetails]:

    def access_detail_builder(access: Access) -> Optional[AccessDetails]:
        try:
            meth, uri, protocol = parse_request(access.request)
            return AccessDetails(
                access=access,
                time=parse_time(access.time),
                method=meth,
                url=urllib.parse.urlparse(uri),
                protocol=protocol,
                referrer=urllib.parse.urlparse(access.referer),
                agent=parse_agent(access.user_agent)
            )
        except ValueError as e:
            print(e, repr(access))
        return None

    return filter(
        None,
        map(access_detail_builder, access_iter)
    )
```

我们从构造 AccessDetails 对象改为构造一个返回单个值的函数。可以将该函数映射到原始 Access 对象的可迭代输入流中，这与 multiprocessing 模块的工作方式非常相适。

在面向对象编程环境中，这些额外的解析器可以是类定义中的方法函数或者属性。具有惰性解析方法的面向对象设计的优点是数据项只在需要时才会被解析。这个特定的函数式设计方法可以解析任何东西，只需假定会用到它。

创建一个惰性函数式设计是可行的，它可以基于 3 个解析器函数并根据需求从 Access 对象中提取并解析各种元素。我们使用 parse_time(access.time)参数而不使用 details.time 属性。尽管语法上变长了，但它确保了只在需要时才解析属性。

12.3.5 过滤访问细节

下面介绍 AccessDetails 对象的几个过滤器。第一个用于排除我们不感兴趣的大量无用文件。第二个过滤器会作为分析函数的一部分，稍后将展开讨论。

函数 path_filter()具有以下 3 个功能：

- 排除空路径；
- 排除特定文件名；
- 排除具有特定扩展名的文件。

优化版的 path_filter()函数如下所示：

```
def path_filter(
        access_details_iter: Iterable[AccessDetails]
    ) -> Iterable[AccessDetails]:
    name_exclude = {
        'favicon.ico', 'robots.txt', 'index.php', 'humans.txt',
        'dompdf.php', 'crossdomain.xml',
        '_images', 'search.html', 'genindex.html',
        'searchindex.js', 'modindex.html', 'py-modindex.html',
    }
    ext_exclude = {
        '.png', '.js', '.css',
    }
    for detail in access_details_iter:
        path = detail.url.path.split('/')
        if not any(path):
            continue
        if any(p in name_exclude for p in path):
            continue
        final = path[-1]
        if any(final.endswith(ext) for ext in ext_exclude):
            continue
        yield detail
```

我们对每个独立的 AccessDetails 对象使用 3 个过滤器测试。如果路径名实际为空，或者含有一个需要排除的名称，又或者路径名称末尾包含需要排除的扩展名，则静默地忽略路径名。如果路径名不满足其中任何一个条件，那么我们可能会对它感兴趣，它也将成为 path_filter() 函数最终结果的一部分。

这是一种优化技术，因为所有测试都使用了命令式的 for 循环体。

另一种设计方法是将每个测试定义为独立的、过滤器风格的头等函数。例如可用以下函数处理空路径：

```
def non_empty_path(detail: AccessDetails) -> bool:
    path = detail.url.path.split('/')
    return any(path)
```

该函数简单地确保了路径中必须包含一个名称。可如下所示使用 filter()函数：

```
filter(non_empty_path, access_details_iter)
```

可以为 non_excluded_names()和 non_excluded_ext()函数编写类似的测试。整个 filter()函数序列如下所示：

```
filter(non_excluded_ext,
    filter(non_excluded_names,
        filter(non_empty_path, access_details_iter)))
```

其中每个 filter()函数都作用于前一个 filter()函数的结果。空路径以及该子集中需要排除的名称和扩展名都会被拒接。还可以将上述示例声明为一系列赋值语句，如下所示：

```
non_empty = filter(non_empty_path, access_details_iter)
nx_name = filter(non_excluded_names, non_empty)
nx_ext = filter(non_excluded_ext, nx_name)
```

该版本的优点是当添加新的过滤条件时易于扩展。



使用生成器函数（如 filter()函数）意味着我们并没有创建大的中间结果对象。每一个中间变量（ne、nx_name 和 nx_ext）都作为合适的惰性生成器函数，不会在客户端进程处理完数据之前执行任何处理。

尽管看上去简单，但由于每个函数都需要解析 AccessDetails 对象中的路径，因此效率很低。为了提高效率，可使用 lru_cache 属性封装一个 path.split('/')函数。

12.3.6　分析访问细节

下面介绍两个分析函数，并使用它们来过滤和分析各个 AccessDetails 对象。第一个 filter()函数只传递特定路径。第二个函数会汇总每个不同路径的出现频次。

下面定义一个小函数 book_in_path()，并结合内置的 filter()函数，用于分析细节。复合的 book_filter()函数如下所示：

```
from typing import Iterable, Iterator
def book_filter(
        access_details_iter: Iterable[AccessDetails]
    ) -> Iterator[AccessDetails]:
    def book_in_path(detail: AccessDetails) -> bool:
        path = tuple(
            item
            for item in detail.url.path.split('/')
            if item
        )
        return path[0] == 'book' and len(path) > 1
    return filter(book_in_path, access_details_iter)
```

这样就通过 book_in_path 函数定义了一条规则，并将其应用于每个 AccessDetails 对象。如果路径名非空，且路径的一级属性是 book，那么我们会对这些对象感兴趣，其他所有 AccessDetails 对象都会被静默排除。

我们最感兴趣的是 reduce_book_total()归约函数，如下所示：

```
from collections import Counter
def reduce_book_total(
        access_details_iter: Iterable[AccessDetails]
    ) -> Dict[str, int]:
    counts: Dict[str, int] = Counter()
    for detail in access_details_iter:
        counts[detail.url.path] += 1
    return counts
```

该函数会生成一个 Counter()对象，用于显示 AccessDetails 对象中每个路径出现的频率。可以使用 reduce_total(book_filter(details))方法关注某组的特定路径，它汇总了给定过滤器可接收的数据项。

由于 Counter 对象适用于各类型，因此需要类型提示来缩小类型范围。在本例中，类型提示是 Dict[str, int]，它会告知 mypy 工具统计路径的字符串表示形式。

12.3.7 完整的分析过程

用于处理日志文件集合的 analysis()复合函数如下所示：

```
def analysis(filename: str) -> Dict[str, int]:
    """Count book chapters in a given log"""
    details = path_filter(
        access_detail_iter(
            access_iter(
                local_gzip(filename))))
```

```
    books = book_filter(details)
    totals = reduce_book_total(books)
    return totals
```

函数 analysis()使用 local_gzip()函数来处理单个文件名或文件模式。它运用一组标准的解析函数 path_filter()、access_detail_iter()和 access_iter()等创建了一个可迭代的 AccessDetails 对象序列。随后它将分析过滤器和归约器应用于该 AccessDetails 对象序列。结果是一个可以显示特定路径访问频次的 Counter 对象。

将这个特定的数据集合保存为了.gzip 格式的日志文件，总数据量约为 51MB。如果用该函数串行处理这些文件，耗时超过 140 秒。能否通过并发处理来获得更好的效果呢?

12.4　使用多进程池进行并发处理

可以使用 multiprocessing 模块创建一个 Pool 处理对象，并将任务分配给进程池中的各个进程。我们将利用操作系统来交错各个进程之间的执行。如果每个进程都混合了 I/O 和计算，那么处理器会处于满负荷工作状态。在进程等待 I/O 完成的同时，其他进程可以执行各自的计算。当 I/O 完成后，进程将启动并与其他进程竞争处理器时间。

将任务映射到独立进程的方法如下所示：

```
import multiprocessing
    with multiprocessing.Pool(4) as workers:
        workers.map(analysis, glob.glob(pattern))
```

上面创建了一个含 4 个独立进程的 Pool 对象，并将这个 Pool 对象赋给了 workers 变量。然后使用该进程池将 analysis 函数映射到了一个待完成的可迭代任务队列。该可迭代队列会为 workers 池中的每个进程分配任务。在本例中，该队列是使用 glob.glob(pattern)属性后得到的，是一个文件名序列。

当 analysis()函数返回结果时，创建 Pool 对象的父进程便可以收集这些结果了。这允许我们创建多个并发构建的 Counter 对象，并将它们合并为单个的复合结果。

如果在进程池中启动 p 个进程，那么整个应用程序将包括 p+1 个进程，由一个父进程和 p 个子进程组成。由于父进程在子进程池启动之后几乎无事可做，因此这种做法通常很有效。通常我们会将 worker 分配至单独的 CPU（或计算核心），而父进程则与 Pool 对象中的一个子进程共享一个 CPU。



常规的 Linux 父/子进程规则适用于由该模块创建的子进程。如果父进程在正确收集到子进程最终状态之前崩溃了，那么便会留下仍在运行的**僵尸**进程。鉴于此，进程的 Pool 对象会作为上下文管理器。当通过 with 语句使用该进程池时，在上下文结束后子进程都会适时终止。

默认情况下，一个 `Pool` 对象会根据 `multiprocessing.cpu_count()`函数的值创建多个 worker。该数值通常是最优的，因此简单使用属性 `multiprocessing.Pool() as workers:` 就足够了。

在某些情况下，创建比 CPU 个数更多的 worker 也会有帮助。当每个 worker 都需要进行 I/O 密集型处理时，这种情况便会存在。让许多 worker 进程等待 I/O 完成可以缩短应用程序的运行时间。

如果给定的 `Pool` 对象中有 p 个 worker，那么这种映射可以将处理器时间几乎缩减至连续处理这些日志所需时间的 $\frac{1}{p}$。实际上，`Pool` 对象中父进程和子进程之间的通信存在一定开销。如果将任务细分为非常小的并发单元，那么这些开销会限制并发的效率。

多进程的 `Pool` 对象有 4 个类 `map()`方法，分别是 `map()`、`imap()`、`imap_unordered()` 和 `starmap()`函数，它们负责在进程池中分配任务。这 4 个函数是一个通用模式的变体，该模式将函数赋给进程池中的每个函数，并将数据映射至该函数。它们分配任务和收集结果的方式不同。

`map(function, iterable)`方法将迭代项分配给进程池中的每个 worker。结果按照分配 `Pool` 对象时的顺序进行收集，以保留原始顺序。

`imap(function, iterable)`方法比 `map()` “懒惰”。默认情况下，它会把每个单独的迭代项发送给下一个可用的 worker。这可能会引入额外的通信开销，因此建议使用大于 1 的块。

`imap_unordered(function, iterable)`方法与 `imap()`方法类似，但它不会保留结果的顺序。允许乱序处理映射意味着一旦进程完成处理便收集结果，否则必须按序收集结果。

`starmap(function, iterable)`方法类似于 `itertools.starmap()`函数。每个迭代项必须是元组，并且为了让元组中的每个值成为位置参数，使用了`*`修饰符将其传递给函数。实际上，它执行的是 `function(*iterable[0])`，`function(*iterable[1])`等函数。

上述映射模式的一个变体如下所示：

```
import multiprocessing
pattern = "*.gz"
combined = Counter()
with multiprocessing.Pool() as workers:
    result_iter = workers.imap_unordered(
        analysis, glob.glob(pattern))
    for result in result_iter:
        combined.update(result)
```

这样就创建了一个 `Counter()`函数，用于合并进程池中每个 worker 的结果。我们基于可用的 CPU 创建了一个子进程池，并使用 `Pool` 对象作为上下文管理器。然后将 `analysis()`函数映射至文件匹配模式中的每个文件。通过 `analysis()`函数得到的所有 `Counter` 对象最终合并成

了一个计数器。

该版本分析一批日志文件花费了大约 68 秒。使用多个并发进程可大幅缩减分析日志的用时。单进程的基准运行时间是 150 秒。如果想确定让系统满负荷运作所需的 worker 数量，需要在更大的进程池中进行额外的试验。

我们使用 multiprocessing 模块中的 Pool.map()函数创建了一个双层“映射–归约”过程。第一层是 analysis()函数，它对单个日志文件进行映射和归约，然后用更高层的归约操作合并这些归约结果。

12.4.1 使用 **apply()**发送单个请求

除了 map()函数的变体，也可以使用 apply(function, *args, **kw)方法向工作池传递值。map()方法实际上只是一个封装了 apply()方法的 for 循环。例如，可以使用以下命令：

```
list(
    workers.apply(analysis, f)
    for f in glob.glob(pattern)
)
```

尚不清楚这样做能否显著带来改进，但可以用 map()函数来表示需要做的几乎任何事。

12.4.2 使用 **map_async()**、**starmap_async()**和 **starmap_async()**等函数

函数 map_async()、starmap_async()和 starmap_async()负责将任务分配给 Pool 对象中的子进程，当子进程完成处理后便从子进程收集结果。这会导致子进程必须等待父进程来收集结果。函数_async()的变体不会等待子进程完成。这些函数会返回一个可供查询的对象，用于从子进程中获取单独的结果。

使用 map_async()方法的一个变体如下：

```
import multiprocessing
pattern = "*.gz"
combined = Counter()
with multiprocessing.Pool() as workers:
    results = workers.map_async(
        analysis, glob.glob(pattern))
    data = results.get()
    for c in data:
        combined.update(c)
```

这样就创建了一个 Counter()函数，用于合并进程池中每个 worker 的结果。我们基于可用的 CPU 创建了一个子进程池，并使用 Pool 对象作为上下文管理器。然后将 analysis()函数映射至文件匹配模式中的每个文件。函数 map_async()返回的是一个 MapResult 对象，可以借此查询结果和工作池的整体状态。这个例子使用了 get()方法来获取 Couter 对象序列。

函数 analysis()生成的 Counter 对象最终组合成了单个 Counter 对象。这种聚合汇总了这些日志文件的所有信息。这种处理方式并不比上一个例子快，但使用 map_async()函数允许父进程在等待子进程完成的同时处理一些额外的工作。

12.4.3 更复杂的多进程架构

multiprocessing 包支持各种架构。我们可以创建跨多个服务器的多进程结构，并提供官方身份认证技术以设立必要的安全等级。可以使用队列和管道将对象从一个进程传递到另一个进程，也可以在进程间共享内存。还可以在进程之间共享底层锁，来同步对共享资源（如文件）访问。

这些架构中，大多数都涉及显式管理多个工作进程的状态，特别是使用锁和共享内存，它们在本质上是命令式的，并不适合函数式编程方法。

在一定程度上，可以用函数式的方法处理队列和管道。我们的目标是将设计分解为生产者函数和消费者函数。生产者函数可以创建对象并将它们插入队列中。消费者函数会从队列中提取对象并进行处理，它可能还会将中间结果放入另一个队列中。这样就创建了一个并发处理网络，其工作负载分布于不同的进程之间。使用 pycsp 包可以简化进程间基于队列的消息交换。更多相关信息，请访问 https://pypi.python.org/pypi/pycsp。

这种设计技术适用于复杂应用程序服务器的设计。各个子进程可以存在于服务器的整个生命周期中，并发处理各自的请求。

12.4.4 使用 concurrent.futures 模块

除了 multiprocessing 包，还可以利用 concurrent.futures 模块。该模块也提供了一种将数据映射到并发的线程池或进程池的方法。该模块的 API 相对简单，并且在许多方面都与 multiprocessing.Pool()函数的接口类似。

12

它们之间的相似度如下所示：

```
from concurrent.futures import ProcessPoolExecutor
pool_size = 4
pattern = "*.gz"
combined = Counter()
with ProcessPoolExecutor(max_workers=pool_size) as workers:
    for result in workers.map(analysis, glob.glob(pattern)):
        combined.update(result)
```

这个例子和先前那些例子之间最主要的不同在于使用了一个 concurrent.futures.ProcessPoolExecutor 对象的实例，而不是 multiprocessing.Pool 方法。这里的基本设计模式是使用可用的工作池将 analysis()函数映射到文件名列表。将生成的 Counter 对象合并

来创建最终结果。

模块 `concurrent.futures` 的性能与 `multiprocessing` 模块几乎相同。

12.4.5　使用 concurrent.futures 线程池

模块 `concurrent.futures` 为应用程序提供了第二种执行器。可以创建并使用 `ThreadPoolExecutor` 对象来代替 `concurrent.futures.ProcessPoolExecutor` 对象，这会在单个进程内创建一个线程池。

线程池的语法与使用 `ProcessPoolExecutor` 对象几乎一致，然而性能却相差很大。在这个分析日志文件的示例中，I/O 占据了主要任务。由于进程中的所有线程都受到统一的操作系统调度限制，因此多线程日志文件分析的整体性能与串行处理日志文件的性能差不多。

使用示例中的日志文件和一台运行 macOS X 的小型四核笔记本电脑，下面的结果显示了共享 I/O 资源的线程与进程之间的性能差异。

- 使用 `concurrent.futures` 线程池，运行时间为 168 秒。
- 使用进程池，运行时间为 68 秒。

在这两种情况下，`Pool` 对象的大小都为 4。单进程和单线程的基准运行时间为 150 秒，引入更多的线程反而拖慢了运行速度。对于执行大量输入和输出的程序来说，这是较为典型的结论。多线程可能更适于处理用户界面：线程长时间处于空闲状态，或者等待用户移动鼠标和触摸屏幕。

12.4.6　使用 threading 模块和 queue 模块

Python 的 `threading` 包有许多支持构建命令式应用程序的构造体。该模块的重点不在于编写函数式应用程序。我们可以利用 `queue` 模块中线程安全的队列将对象从一个线程传递到另一个线程。

模块 `threading` 中并没有简单的方法能将任务分配至各个线程，其 API 并不太适合函数式编程。

与 `multiprocessing` 模块中更为原始的特性一样，可以尝试隐藏锁与队列的状态特性和命令特性。然而，使用 `concurrent.futures` 模块中的 `ThreadPoolExecutor` 方法似乎更简便。`ProcessPoolExecutor.map()` 方法有一个非常易用的接口，可用于并发处理集合元素。

使用 `map()` 函数原语分配任务似乎很符合函数式编程的要求，因此要多关注 `concurrent.futures` 模块，它最适于编写并发函数式应用程序。

12.4.7 设计并发处理

从函数式编程的角度看,可以使用以下 3 种方法将map()函数的概念应用于数据项的并发处理:

- multiprocessing.Pool
- concurrent.futures.ProcessPoolExecutor
- concurrent.futures.ThreadPoolExecutor

这些方法同我们与之交互的方式几乎相同,因为这 3 种方法都有一个 map()方法将某个函数应用于可迭代集合数据项,这与其他函数式编程技术非常相称。并发线程和并发进程因性质不同,性能也有所不同。

随着设计逐步完善，日志分析应用程序可分为以下两大块。

- 底层解析：几乎所有日志分析应用程序都可以使用的通用解析。
- 高层分析应用：针对具体应用程序需求的过滤和归约处理。

底层解析可以分解为如下 4 个阶段。

- 从多个源日志文件读取所有行数据。这是从文件名到行序列的 local_gzip()映射。
- 通过文件集合中每一行日志条目创建的简单命名元组。这是从文本行到 Access 对象的 access_iter()映射。
- 解析更复杂字段（如日期和 URL）的详细信息。这是从 Access 对象到 AccessDetails 对象的 access_detail_iter()映射。
- 从日志中排除不感兴趣的路径名。也可以将其视作只接收感兴趣的路径名。与其说是映射操作，它更像是过滤器，一个捆绑成 path_filter()函数的过滤器集合。

前面定义了一个完整的 analysis()函数来解析和分析给定的日志文件。该函数使用高层的过滤器和归约器来处理底层的解析结果，同时它也适用于使用通配符的文件集合。

考虑到所涉及的映射个数，有多种方法能将该问题分解为可以映射到线程池或进程池的任务。可以考虑将以下映射方法作为设计备选方案。

- 将 analysis()函数映射至各个文件。本章示例统一使用了该方法。
- 从整个 analysis()函数中重构出 local_gzip()函数,就可以将修改后的 analysis()函数映射到 local_gzip()函数的结果上。
- 从整个 analysis()函数中重构出 access_iter(local_gzip(pattern))函数，就可以将修改后的 analysis()函数映射到可迭代的 Access 对象序列上。
- 将 access_detail_iter(access-iter(local_gzip(pattern)))函数重构为单独的迭代函数，然后将 path_filter()函数与高层的过滤器和归约器映射到可迭代的 AccessDetail 对象序列上。

12

- 还可以将底层解析重构为与高层分析相分离的函数，将分析过滤器和归约器映射到底层解析的输出上。

所有这些用于重构示例应用程序的方法都相对比较简单。运用函数式编程技术的好处是整个过程中的每个部分都可以定义为一个映射。这在考察不同架构以确定最佳设计时非常实用。

但是在这种情况下，需要将 I/O 处理尽量分配给更多的 CPU 或计算核心。大部分可行的重构方法都会在父进程中执行所有 I/O，而只将计算部分分配给多个并发进程，所以不会带来什么益处，因此采用尽可能将 I/O 分配到更多计算核心的映射方法。

最小化进程之间传递的数据量通常是很重要的。在这个例子中，我们只提供给每个工作进程以简短的文件名字符串。相比压缩每个日志文件得到的 10MB 数据详情，生成的 `Counter` 对象要小得多。通过剔除只出现一次的数据项，或者限制应用程序只保留频率最高的 20 项，便可以进一步减小每个 `Counter` 对象。

可以自由地重新组织这个应用程序的设计并不意味着应该这么做。我们可以运行一些基准测试实验来验证猜测，即日志文件的解析主要是由读取文件的时间决定的。

12.5　小结

本章介绍了支持多个数据并发处理的两种方法。

- 使用 `multiprocessing` 模块：具体而言，是 `Pool` 类和适用于工作池的各种映射机制。
- 使用 `concurrent.futures` 模块：具体而言，是 `ProcessPoolExecutor` 类和 `ThreadPoolExecutor` 类。这些类也支持能在工作线程间和工作进程间分配任务的映射机制。

本章还提到了一些似乎不太适合函数式编程的替代方案。模块 `multiprocessing` 有其他许多特性，但并不太适合函数式设计。类似地，可以用 `threading` 模块和 `queue` 模块构建多线程应用程序，但它们的特性也使其不适合函数式程序。

下一章将介绍 `operator` 模块，可用该模块简化一些算法。我们可以使用内置的运算符函数而不是定义一个匿名函数。下一章还将介绍一些制定灵活决策的技术，这些技术还允许以非严格顺序对表达式进行求值。

第 13 章

条件表达式和 operator 模块

函数式编程强调惰性求值和按非严格顺序执行运算，其思想是允许编译器或运行时环境以尽量少的工作计算结果。Python 则倾向于施加严格的求值顺序，而这可能会导致低效。

Python 中的 `if`、`elif` 和 `else` 语句对条件求值施以严格的执行顺序。本章会介绍如何摆脱这种严格的求值顺序，并在有限范围内写出非严格的条件语句。尚不清楚这样做能否有帮助，但它展示了如何以更函数式的方法表示算法。

前面介绍了一些高阶函数。在某些情况下，可以使用这些高阶函数将一些复杂的函数应用于数据集合，而在其他一些情况下，可以使用简单函数处理数据集合。

事实上，在许多情况下可以编写小型匿名函数对象，以便将一个 Python 运算符应用于某个函数，例如可以使用以下代码定义 `prod()` 函数：

```
from typing import Iterable
from functools import reduce
def prod(data: Iterable[int]) -> int:
    return reduce(lambda x, y: x*y, data, 1)
```

相较于简单的乘法，使用参数 `lambda x，y: x*y` 似乎有些累赘，毕竟我们只想使用乘法运算符（`*`）。能否简化语法呢？答案是肯定的，因为 `operator` 模块提供了内置运算符的定义。在示例中可以用 `operator.mul` 代替匿名函数对象。

本章将讨论以下主题。

- 如何实现非严格求值。介绍的工具很有用，可以优化性能。
- `operator` 模块，以及它是如何为创建高阶函数实施简化和明晰化的。
- 星号映射，即用 `f(*args)` 提供多参数的映射。
- 一些更高级的 `partial()` 和 `reduece()` 技术。

13.1　条件表达式求值

Python 对表达式执行严格排序，其中值得注意的例外情况是短路运算符 and 和 or。对语句的求值也有严格的顺序要求，这使得难以对其进行优化，因为可能会破坏严格的求值顺序。

条件表达式的求值让我们可以尝试非严格语句执行顺序。Python 中的 if、elif 和 else 语句会严格按照从头至尾的顺序进行求值。理想情况下，一门可优化的语言可能会放宽这条规则，这样编译器就可以发现更快的运算顺序来求解条件表达式了，这让我们可以编写出对读者来说顺序合理的表达式，并让编译器能找到更快的求值顺序。

由于缺少优化编译器，非严格执行顺序的概念对 Python 来说有点牵强。尽管如此，确有其他办法可以将条件语句表示为函数的求值而非执行命令式语句，以此重排运行时语句。

Python 具有 if 和 else 条件表达式。运算符 if-else 是一个短路运算符。这会带来一些很细微的优化，因为两个外部条件中只有一个会基于内部条件的真实性进行求值。当写下 x if c else y 时，只有在 c 为 True 时才会求解表达式 x。此外，表达式 y 也只有在 c 为 False 时才会求值。这是一个很细微的优化，但仍严格执行运算顺序。

该表达式对于简单条件来说很有用，然而当有多个条件时，它会变得非常复杂，需要小心翼翼地嵌套子表达式，甚至最后可能会得到晦涩难懂的表达式，如下所示：

```
(x if n==1 else (y if n==2 else z))
```

上述表达式只会对 x、y 和 z 中的一个进行求值，具体取决于 n 的值。

如果研究 if 语句，会发现可以用一些数据结构模拟出 if 语句的效果。其中一种技术是使用字典的键和匿名函数对象来创建条件和值的一组映射。用表达式表示阶乘函数的一种方法如下：

```
def fact(n: int) ->int:
    f = {
        n == 0: lambda n: 1,
        n == 1: lambda n: 1,
        n == 2: lambda n: 2,
        n > 2: lambda n: fact(n-1)*n
    }[True]
    return f(n)
```

该表达式会将传统的 if、elif 和 else 语句序列重写为单个表达式。为了更清晰地展示其内部机制，将其分为以下两个步骤。

第一步对各个条件进行求值。如果其中一个给定条件为 True，则其余条件都为 False。生成的字典将包含两项：一个键为 True 的匿名函数对象和一个键为 False 的匿名函数对象。我们会选取键为 True 的项并将其赋给变量 f。

该映射过程中使用了匿名函数作为键值，这样在构建字典时并不会对表达式进行求值。我们

希望字典选取其中一个匿名函数，并且匿名函数的值便是整个函数的结果。对于输入的参数 `n`，`return` 语句求解了一个条件为 `True` 的匿名函数 `f`。

13.1.1　使用非严格字典规则

Python 3.6 之前没有定义字典键的顺序。如果试图创建一个包含多个项的字典，且这些项共享同一个键值，那么最终生成的 `dict` 对象只会保留其中一项。我们无法定义被保留的是这些重复键值中的哪一个，但只要设计的算法是正确的，这一点便无关紧要。

典型结果如下所示。最后一个值会替换任何先前的值。但在 Python 3.6 之前，无法保证这种情况一定会发生。

```
>>> {'a': 1, 'a': 2}
{'a': 2}
```

在这种情况下，无须在意需要保留哪一个重复键。下面是 `max()` 函数的一个简化版本，它简单地选取了两个值中较大的那个。

```
def non_strict_max(a, b):
    f = {a >= b: lambda: a,
         b >= a: lambda: b}[True]
    return f()
```

对于 `a == b` 的情况，字典中的两项都会得到条件为 `True` 的键，但其中只有一个会保留下来。由于这两个结果一样，因此保留哪个或将哪个视为重复并进行覆盖并不重要。

请注意，该函数的正规类型提示非常复杂。进行比较的项必须是**可排序的**，即它们必须将运算符排序。下面定义一个适用于可排序对象概念的类型。

```
from abc import ABCMeta, abstractmethod
from typing import TypeVar, Any

class Rankable(metaclass=ABCMeta):
    @abstractmethod
    def __lt__(self, other: Any) -> bool: ...
    @abstractmethod
    def __gt__(self, other: Any) -> bool: ...
    @abstractmethod
    def __le__(self, other: Any) -> bool: ...
    @abstractmethod
    def __ge__(self, other: Any) -> bool: ...

RT = TypeVar('RT', bound=Rankable)
```

类 `Rankable` 的定义是一个抽象类。针对某些有用的函数定义，这个类借助 `abc` 模块来将类定义中的抽象特性具象化。装饰器`@abstractmethod` 用于确定任何具体且实用的子类必须定义的方法函数。

之后可以将 non_strict_max()函数的参数类型和返回值类型绑定为类型变量 RT。包含类型提示的定义如下所示（此处省略了函数主体，上文已经给出了）：

```
def non_strict_max(a: RT, b: RT) -> RT:
```

这里阐明了接收的两个参数 a 和 b 应是可排序的类型，并被指定为了 RT 类型。返回值的可排序类型相同。这样就厘清了 max()的基本语义，即参数的类型必须一致，返回值的类型也和参数的类型相同。

13.1.2　过滤 **True** 条件表达式

确定哪个表达式的结果为 True 有多种方法。前面的示例将键加载到了字典中。这种字典加载方法只会保留一个键为 True 的值。

针对这个模型，使用 filter()函数编写的另一种变体如下所示：

```
from operator import itemgetter
def semifact(n: int) -> int:
    alternatives = [
        (n == 0, lambda n: 1),
        (n == 1, lambda n: 1),
        (n == 2, lambda n: 2),
        (n > 2, lambda n: semifact(n-2)*n)
    ]
    _, f = next(filter(itemgetter(0), alternatives))
    return f(n)
```

这样就将所有备选方案定义为条件和函数对的一组序列，其中每一项都作为一个条件，且其基于输入和能生成输出结果的匿名函数。变量赋值语句中还可以包含一个类型提示，如下所示：

```
alternatives: List[Tuple[bool, Callable[[int], int]]] = [
    etc,
]
```

这个列表实际上代表的是 4 个二元组的同一集合。该定义阐明了其中的元组列表包含一个布尔值和一个可调用函数。

当使用 itemgetter(0)参数来应用 filter()函数时，我们会选取元组第 0 项值为 True 的对。对于这些值为 True 的对，使用 next()方法从 filter()函数创建的可迭代对象中提取第一项。选取的条件值赋给_变量，选取的函数则赋给 f 变量。可以忽略条件值（它为 True），而只对返回的 f()函数进行求值。

前面的示例，使用匿名函数将函数的求值延迟到了条件求值之后。

其中的 semifact()函数也称**双阶乘**[①]。半阶乘的定义和阶乘类似，主要的区别在于半阶乘

① 或“半阶乘”。——译者注

不是所有数字的乘积，而是交替数字的乘积。例如下面两个公式：

- 5!!=5×3×1
- 7!!=7×5×3×1

13.1.3　寻找匹配模式

上述这种创建多个条件集合的技术也可以配合正则表达式使用。回想一下，模式的 match() 方法或 search()方法会返回一个匹配对象或者 None 值。可以在程序中利用这一点，如下所示：

```
import re
p1 = re.compile(r"(some) pattern")
p2 = re.compile(r"a (different) pattern")

from typing import Optional, Match
def matcher(text: str) -> Optional[Match[str]]:
    patterns = [p1, p2]
    matching = (p.search(text) for p in patterns)
    try:
        good = next(filter(None, matching))
        return good
    except StopIteration:
        pass
```

上面定义了两种模式，并将它们应用于给定的文本块。每个模式都有一个用()标记的子模式来作为模式匹配组。

函数 matcher()会构建一系列可选模式。在本例中，它是一组简单文本对的模式。我们用生成器表达式将每个模式中的 search()方法应用于所提供的文本。由于生成器表达式是惰性求值的，因此它不会即刻执行一长串连续的模式匹配，而会先使用已得到的结果。

以 None 作为第一个参数的 filter()函数会从数据序列中去除所有的 None 值。filter(None, S)得到的值与 filter(lambda item: item is not None, S)得到的值相同。

函数 next()会从函数 filter()返回的可迭代结果中获取第一个非空值。如果 filter()函数没有返回任何结果，则表示没有与之匹配的模式。在这种情况下，异常值转换为了 None 结果。由于没有与给定文本匹配的模式，因此引发一个自定义的异常或许是明智的做法。

与先前的示例一样，这里展示了如何对多个布尔条件进行求值并从中选取一个真值。由于输入的是一个模式序列，因此不同函数的求值顺序是定义好的，并且严格按照顺序执行。尽管无法摆脱 Python 的严格求值顺序，但是一旦找到某个匹配的模式，便可以立即退出函数来控制求值的成本。

13.2　使用 operator 模块代替匿名函数

在使用 max()、min()和 sorted()这些函数时，用到了一个可选的参数 key=。作为参数值的函数修改了高阶函数的行为。在许多情况下，可使用简单的匿名函数从元组中选取数据项，见以下两个典型示例：

```
from typing import Callable, Sequence, TypeVar
T_ = TypeVar("T_")
fst: Callable[[Sequence[T_]], T_] = lambda x: x[0]
snd: Callable[[Sequence[T_]], T_] = lambda x: x[1]
```

它们与其他一些函数式编程语言中的内置函数相匹配，用于选取元组的第一项或第二项。这里使用了类型提示来避免发生其他类型的转换，即把序列中项的类型绑定为类型变量 T_，以此表示函数返回结果的类型。

其实无须编写这样的函数，可以使用 operator 模块中 itemgetter()的函数版本。它们都是高阶函数，即表达式 itemgetter(0)会创建一个函数。随后可以应用该函数来选取集合中的某个对象，如下所示：

```
>>> from operator import itemgetter
>>> itemgetter(0)([1, 2, 3])
1
```

下面在一个更复杂的元组列表数据结构中使用该函数。示例数据如下：

```
year_cheese = [
    (2000, 29.87), (2001, 30.12), (2002, 30.6), (2003, 30.66),
    (2004, 31.33), (2005, 32.62), (2006, 32.73), (2007, 33.5),
    (2008, 32.84), (2009, 33.02), (2010, 32.92)
]
```

以上数据代表每年的奶酪消耗量。第 2 章和第 9 章用过这个示例。

可以使用以下命令来确定最小奶酪量的数据点：

```
>>> min(year_cheese, key=snd)
(2000, 29.87)
```

模块 operator 提供了一种从元组中获取特定元素的替代方法。这使我们能免于使用匿名函数变量来选取其中第二项。

如以下命令所示，无须自定义 fst()函数和 snd()函数，而可以使用参数 itemgetter(0)和 itemgetter(1)。

```
>>> from operator import itemgetter
>>> max(year_cheese, key=itemgetter(1))
(2007, 33.5)
```

函数 itemgetter()依赖一个特殊方法__getitem__()，它会根据数据项在元组（或列表）中的索引位置来获取其值。

使用高阶函数获取命名属性

下面来看稍微不同的数据集合。假设这里使用的是 NamedTuple 的子类而非匿名元组对象。首先定义一个类，它为元组中的两项提供了以下类型提示。

```
from typing import NamedTuple
class YearCheese(NamedTuple):
    year: int
    cheese: float
```

随后将基础数据 year_cheese 转换为合适的命名元组。转换过程如下所示：

```
>>> year_cheese_2 = list(YearCheese(*yc) for yc in year_cheese)

>>> year_cheese_2
[YearCheese(year=2000, cheese=29.87),
 YearCheese(year=2001, cheese=30.12),
 YearCheese(year=2002, cheese=30.6),
 YearCheese(year=2003, cheese=30.66),
 YearCheese(year=2004, cheese=31.33),
 YearCheese(year=2005, cheese=32.62),
 YearCheese(year=2006, cheese=32.73),
 YearCheese(year=2007, cheese=33.5),
 YearCheese(year=2008, cheese=32.84),
 YearCheese(year=2009, cheese=33.02),
 YearCheese(year=2010, cheese=32.92)]
```

有两种方法来确定奶酪消耗量的范围，可以如下所示使用 attrgetter()函数，或者使用匿名函数形式。

```
>>> from operator import attrgetter
>>> min(year_cheese_2, key=attrgetter('cheese'))
YearCheese(year=2000, cheese=29.87)
>>> max(year_cheese_2, key=lambda x: x.cheese)
YearCheese(year=2007, cheese=33.5)
```

这里的重点是，对于匿名函数对象，属性名是代码中的一个标记，而对于 attrgetter()函数，属性名是一个字符串。使用字符串的话，属性名会是一个在脚本运行期间可以修改的参数，这会使程序更为灵活。

13.3 运算符的星号映射

函数 itertools.starmap()是高阶函数 map()的一个变体。函数 map()会将某个函数应用于序列中的每一项。函数 starmap(f, S)则假定序列 S 中的每一项 i 都是一个元组，并且使

用 f(*i)作为映射。每个元组中数据项的个数必须与给定函数中的参数个数相匹配。

示例如下：

```
>>> d = starmap(pow, zip_longest([], range(4), fillvalue=60))
```

函数 itertools.zip_longest()会创建如下一组序列对：

```
[(60, 0), (60, 1), (60, 2), (60, 3)]
```

该函数之所以会得到此结果是因为我们提供了两个序列：方括号[]和参数 range(4)。当较短序列中的数据处理完后，会使用 fillvalue 参数。

当使用 starmap()函数时，每一个元组对都会作为给定函数的参数。本例中使用的是 operator.pow()函数，即**运算符。该表达式计算了[60**0, 60**1, 60**2, 60**3]的值。变量 d 的值是[1, 60, 3600, 216000]。

函数 starmap()接收元组序列。map(f, x, y)函数和 starmap(f, zip(x, y))函数大致等同。

上述示例中 itertools.starmap()函数的延续如下所示：

```
>>> p = (3, 8, 29, 44)
>>> pi = sum(starmap(truediv, zip(p, d)))
```

上面打包了分别含 4 个值的两个序列。变量 d 的值由上述的 starmap()计算得到，变量 p 代表一个简单的文本项列表，我们将这两个变量打包成对。starmap()函数与 operator.truediv()函数配合使用，后者即/运算符。

这样就能对一组分数序列进行求和了，其结果约为：$\pi \approx \frac{3}{60^0}+\frac{8}{60^1}+\frac{29}{60^2}+\frac{44}{60^3}$。

一个更简化的版本如下所示，它使用了 map(f, x, y)函数代替 starmap(f, zip(x,y))函数。

```
>>> pi = sum(map(truediv, p, d))
>>> pi
3.1415925925925925
```

在本例中，基数为 60 的分数转换为了基数为 10 的分数。变量 d 中的值序列会作为合适的分母。也可以用类似于前面介绍的技术来转换其他基数。

有些近似可能涉及无穷项的求和（或乘积）。可以用类似于前面介绍的技术对其进行求值。可以利用 itertools 模块中的 count()函数来近似生成任意数量的项。随后可以使用 takewhile()函数只累计那些有助于提高解的精确度的值。从另一个角度来看，takewhile()生成了一个显著值的数据流，并且会在遇到非显著值的时候停止处理。

计算一个潜在无穷序列之和的示例如下：

```
>>> from itertools import count, takewhile
>>> num = map(fact, count())
>>> den = map(semifact, (2*n+1 for n in count()))
>>> terms = takewhile(
... lambda t: t > 1E-10, map(truediv, num, den))
>>> 2*sum(terms)
3.1415926533011587
```

变量 num 是一个潜在的无穷分子序列，且它基于前面示例中定义的阶乘函数。函数 count() 返回升序的值，即从零开始无限递增。变量 den 是一个潜在的无穷分母序列，且它基于半阶乘函数。前面的示例中定义过这个函数，它也使用 count()来创建一个潜在的无穷值序列。

通过 map()函数将 operators.truediv()函数（即/运算符）应用于每一对值来创建数据项。将其封装在 takewhile()函数中，以便只在数据值大于某个相对较小的值时（本例中为 10^{-10}），才从 map()的输出中提取该项数据。

该定义的级数展开式如下所示：

$$4\arctan(1)=\pi=2\sum_{n=0}^{\infty}\frac{n!}{(2n+1)!!}$$

这个级数展开式的一个有趣的变体是将 operator.truediv()函数替换为 fractions.Fraction()函数，这会创建不受浮点数近似限制的精确有理值。

可以使用 operators 模块中的所有 Python 内置运算符，其中包括所有位处理运算符以及比较运算符。在某些情况下，相比形式复杂的用函数表示运算符的 starmap()函数，生成器表达式更为简洁明了。

模块 operator 的作用是简化匿名函数。可以使用 operator.add 方法代替 add=lambda a, b: a+b 方法。如果表达式比单个运算符复杂，那么唯一的方法便是编写匿名函数。

13.4 使用 operator 模块函数进行归约

下面再介绍一种使用运算符定义的方法，可以配合内置的 functools.reduce()函数一起使用，例如 sum()函数定义如下：

```
sum = functools.partial(functools.reduce, operator.add)
```

这会根据提供的第一个参数创建一个部分求值的 reduce()函数版本。在本例中，该参数是+运算符，并且由 operator.add()函数实现。

如果需要一个类似的函数来计算乘积，那么可以如下定义：

```
prod = functools.partial(functools.reduce, operator.mul)
```

这里遵循了前面例子中的模式，创建了一个以*运算符作为第一个参数并且进行部分求值的 reduce()函数。

尚不清楚能否用其他运算符来实现类似的功能。我们可能会找到 operator.concat()函数、operator.and()函数和 operator.or()函数的类似实现。



函数 and()和 or()就是按位的&和|运算符。如果需要正确使用布尔运算符，那么必须使用 all()函数和 any()函数代替 reduce()函数。

一旦有了 prod()函数，便意味着可以定义阶乘，如下所示：

```
fact = lambda n: 1 if n < 2 else n*prod(range(1, n))
```

这样做的优点是代码简洁，因为实现了阶乘的单行定义。它还具有不依赖递归的优点，并且避免了因为栈限制而出现的任何问题。

尚不清楚与 Python 中的许多替代方案相比，这样做是否有任何明显优势。从 partial()函数、reduce()函数和 operator 模块等原语片段构建复杂函数的概念是很美妙的。然而在大多数情况下，operator 模块中的简单函数用处不大，而我们总希望使用更复杂的匿名函数。

对于布尔归约，必须使用内置的 any()函数和 all()函数，它们实现了一种短路的 reduce()运算。它们不是高阶函数，而且必须配合匿名函数或已定义的函数使用。

13.5　小结

本章介绍了 if、elif 和 else 语句序列的不同实现版本。理想情况下，使用条件表达式可以执行一些优化操作。实际上，Python 本身不会进行优化，因此使用更奇特的方法来处理条件表达式并不会带来实质性的益处。

本章还介绍了如何使用 operator 模块中的高阶函数，例如 max()、min()、sorted()和 reduce()等。使用运算符可以让我们免于创建许多小型匿名函数。

下一章将介绍 PyMonad 库，它直观地展现了 Python 函数式编程的概念。由于 Python 底层是一种命令式编程语言，因此通常无须使用单子。

与状态变量的赋值相比，有些算法用单子表示会更清晰。届时会展示一个使用单子简洁地表达一组复杂规则的示例。最重要的是，operator 模块展示了许多函数式编程技术。

第 14 章 PyMonad 库

14

利用单子，我们能以一种宽松的语言形式来指定表达式的求值顺序，例如可以使用单子来要求表达式 $a+b+c$ 严格按照从左到右的顺序进行求值。这可能会干扰编译器优化表达式求值的能力，然而有时有必要这样做，例如希望以特定的顺序读写文件，此时单子可以确保按照指定的顺序执行 `read()`函数和 `write()`函数。

语法宽松且有编译器优化的语言因单子指定了表达式的求值顺序而获益。Python 在很大程度上是语法严格且不含优化的，因此对单子没有实际的需求。

然而，PyMonad 包不只包含单子，还有许多具有独特实现的函数式编程特性。在某些情况下，使用 PyMonad 模块编写的程序比仅使用标准库模块编写的程序更加简洁明了。

本章将介绍以下内容：

- 下载并安装 PyMonad；
- 柯里化概念以及如何将其运用于函数式复合；
- 用于创建复合函数的 PyMonad*运算符；
- 函子和使用更通用的函数柯里化数据项的技术；
- `bind()`运算使用>>运算符创建有序单子；
- 如何运用 Pymonad 技术构建蒙特卡洛模拟。

14.1　下载和安装

根据 Python 包索引（PyPI）可以找到 PyMonad 包。使用 `pip` 将 PyMonad 加入运行环境。

访问 https://pypi.python.org/pypi/PyMonad 以获取更多信息。

对于 macOS 和 Linux 开发人员，命令 `pip install pymonad` 可能需要以 `sudo` 命令为前缀。如果你安装的是个人版的 Python，那么无须使用 `sudo`。如果你安装的是系统级的 Python，那么需要使用 `sudo`。在执行 `sudo pip install pymonad` 这样的命令时，系统会提示输入密

码，以确保你拥有执行安装所需的管理权限。对于 Windows 开发人员来说，不存在 sudo 命令，但也必须拥有足够的管理权限。

安装完 PyMonad 包后，可以使用以下命令进行确认：

```
>>> import pymonad
>>> help(pymonad)
```

该命令会显示 docstring 模块，确认已正确安装 PyMonad 包。

整个项目的名称 PyMonad 使用了大小写混合的形式。在导入已安装的 Python 包名“PyMonad”时，使用的都是小写。目前，PyMonad 包中还没有类型提示。通用化的本质要求尽量使用 TypeVar 提示来描述各个函数的签名。此外，PyMonad 中有一些名称与内置 typing 模块中的名称相冲突。由于需要使用包名来消除由名称覆盖引起的歧义，因此类型提示的语法可能会变得非常复杂。本章的示例将忽略类型提示。

14.2　函数式复合和柯里化

一些函数式语言的工作原理是将多参数函数语法转化为单参数函数集合，这一过程称为**柯里化**，它是以逻辑学家 Haskell Curry 的名字命名的。Haskell Curry 从早期概念中发展出了该理论。

柯里化是一种将多参数函数转化为单参数高阶函数的技术。例如函数 $f(x,y)\rightarrow z$，对于给定的两个参数 x 和 y，该函数会返回 z 值作为结果。可以将 $f(x,y)$柯里化为两个函数：$f_{c1}(x)\rightarrow f_{c2}(y)$和 $f_{c2}(y)\rightarrow z$。对于给定的第一个参数值 x，求解函数 $f_{c1}(x)$会返回一个新的单参数函数 $f_{c2}(y)$。第二个函数会接收第二个参数值 y，并返回结果 z。

可以在 Python 中求解一个柯里化函数：f_c(2)(3)。对第一个参数值 2 使用柯里化函数，创建一个新函数。随后对第二个参数值 3 使用这个新函数。

这种做法适用于任何复杂的函数。如果从函数 $g(a,b,c)\rightarrow z$ 开始，那么可以将其柯里化为函数 $g_{c1}(a)\rightarrow g_{c2}(b)\rightarrow g_{c3}(c)\rightarrow z$，这是通过递归实现的。首先，对函数 $g_{c1}(a)$求值会返回一个带参数 b 和 c 的新函 $g'_{c1}(b,c)$数。然后柯里化这个返回的双参数函数并创建函数 $g_{c2}(b)\rightarrow g_{c3}(c)$。

可以用 g_c(1)(2)(3)来求解一个复杂的柯里化函数。这种语法形式过于庞大，因此可以使用一些语法糖来将 g_c(1)(2)(3)缩减为 g(1, 2, 3)这样更容易接受的形式。

下面给出一个 Python 中的具体示例。例如有一个形式如下的函数：

```
from pymonad import curry
@curry
def systolic_bp(bmi, age, gender_male, treatment):
    return (
        68.15+0.58*bmi+0.65*age+0.94*gender_male+6.44*treatment
    )
```

这是一个简单的基于多元回归的心脏收缩压模型。它可以根据**体重指数**（BMI）、年龄、性别（1 代表男性）和既往治疗史（1 代表有过治疗）来预测血压。有关该模型及其推导过程的更多信息，请访问：http://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/BS704_Multivariable/BS704_Multivariable7.html。

可以使用 `systolic_bp()`函数处理所有 4 个参数，如下所示：

```
>>> systolic_bp(25, 50, 1, 0)
116.09
>>> systolic_bp(25, 50, 0, 1)
121.59
```

一位 BMI 为 25，且此前没有接受过治疗的 50 岁男性，其血压会在 116 左右。第二个示例显示了一位与之类似但接受过治疗的女性，其血压可能在 121 左右。

由于使用了`@curry` 装饰器，因此可以得到与函数部分求值类似的中间结果。见如下命令片段：

```
>>> treated = systolic_bp(25, 50, 0)
>>> treated(0)
115.15
>>> treated(1)
121.59
```

上述示例中，通过对 `systolic_bp(25, 50, 0)`方法进行求值创建了一个柯里化函数，并将其赋给了 `treated` 变量。对于给定的患者，BMI、年龄和性别的值通常不会改变。现在可以对剩余的参数使用新的 `treated` 函数，根据患者的病史获取不同的血压预测值。

这在某些方面类似于 `functools.partial()`函数。重要的区别在于柯里化创建了一个能以多种方式运作的函数，而 `functools.partial()`函数则会创建一个更具针对性的函数，并且只能用于专门的数值集合。

创建其他一些柯里化函数的示例如下：

```
>>> g_t= systolic_bp(25, 50)
>>> g_t(1, 0)
116.09
>>> g_t(0, 1)
121.59
```

这是一个基于初始模型且区分性别的函数。必须提供性别和过往治疗值，以便从模型中得到最终结果。

14.2.1 使用柯里化的高阶函数

尽管可以使用普通函数表现柯里化，但只有将柯里化用于高阶函数时，其价值才会真正显现。在理想情况下，`functools.reduce()`函数是可柯里化的，因此可以如下操作：

```
sum = reduce(operator.add)
prod = reduce(operator.mul)
```

然而这样做并不起作用。内置的 reduce()函数不能经由 PyMonad 库实现柯里化，因此上面的例子实际上无法正常运作。但是，如果自定义了 reduce()函数，那么就可以如前所示对其进行柯里化了。

下面是一个自定义的 reduce()函数，可以如前所述使用它：

```
from collections.abc import Sequence
from pymonad import curry

@curry
def myreduce(function, iterable_or_sequence):
    if isinstance(iterable_or_sequence, Sequence):
        iterator= iter(iterable_or_sequence)
    else:
        iterator= iterable_or_sequence
    s = next(iterator)
    for v in iterator:
        s = function(s,v)
    return s
```

函数 myreduce()的行为和内置的 reduce()函数类似。函数 myreduce()适用于可迭代对象或者序列对象。对于给定序列，可以创建一个迭代器，而对于给定的可迭代对象，可以直接使用它。使用迭代器中的第一项来初始化结果。将该函数应用于进行中的求和（或乘积）以及随后的每一项。



还可以封装内置的 reduce()函数来创建一个可柯里化的版本。只需两行代码便可以实现，这留给读者作为练习。

由于 myreduce()函数是一个柯里化函数，因此接下来可以基于该高阶函数创建函数。

```
>>> from operator import add
>>> sum = myreduce(add)
>>> sum([1,2,3])
6
>>> max = myreduce(lambda x,y: x if x > y else y)
>>> max([2,5,3])
5
```

我们通过应用于 add 运算符的柯里化归约自定义了 sum()函数，还使用选取两个值中较大值的匿名函数对象自定义了默认的 max()函数。

由于柯里化关注位置参数，因此无法以这种方式简单地创建出 max()函数的更通用的形式。如果使用 key=关键字参数，那么为了实现函数式编程简洁明了的整体目标，会在技术上带来极大的复杂性。

为了创建一个更通用的 max()函数，需要摆脱函数 max()、min()和 sorted()等依赖的 key=关键字参数范式。必须像 filter()、map()和 reduce()函数那样以高阶函数作为第一个参数。还可以创建一个更统一的高阶柯里化函数库，这些函数将完全依赖位置参数。首先以高阶函数为参数，这样自定义的柯里化 max(function, iterable)方法就能遵循 map()、filter()和 functools.reduce()等函数设置的模式了。

14.2.2 避易就难的柯里化

可以不用 PyMonad 库的装饰器而手动创建柯里化函数。通过函数定义的形式实现的一种方法如下所示：

```
def f(x, *args):
    def f1(y, *args):
        def f2(z):
            return (x+y)*z
        if args:
            return f2(*args)
        return f2
    if args:
        return f1(*args)
    return f1
```

这里将函数 $f(x,y,z) \rightarrow (x+y)*z$ 柯里化为了函数 f(x)，它会返回一个函数，从概念上讲就是 $f(x) \rightarrow f'(y, z)$。然后柯里化中间函数并创建了函数 f1(y)和 f2(z)，即 $f'(y,z) \rightarrow (f_2(y) \rightarrow f_3(z))$。

对 f(x)函数进行求值会得到一个新函数 f1。如果提供了其他参数，则这些参数会传递给 f1 函数用于求值，其结果可以是一个最终值，也可以是另一个函数。

显然，这种通过手动扩展实现柯里化的做法很容易出错，并不是处理函数的实用方法，但是可以用它来说明柯里化的含义以及它在 Python 中的实现方式。

14.3 函数式复合和 PyMonad*运算符

柯里化函数的一个重要价值是可以通过函数式复合来组合函数。第 5 章和第 11 章介绍了函数式复合。

创建了一个柯里化函数后，就可以更轻松地通过函数式复合来创建新的、更复杂的柯里化函数了。例如 PyMonad 包定义了*运算符来复合两个函数。为了解释这个运算符的工作原理，下面定义两个用于复合的柯里化函数。首先定义一个用于计算乘积的函数，然后定义一个用于计算特定范围值的函数。

用于计算乘积的第一个函数如下所示：

14

```
import  operator
prod = myreduce(operator.mul)
```

该函数基于此前定义的柯里化函数 myreduce()。它使用 operator.mul()函数来计算一个可迭代对象的**乘法归约**，换言之，将乘积定义为了一组序列的乘法归约。

用于生成一系列数值的第二个柯里化函数如下所示：

```
@curry
def alt_range(n):
    if n == 0:
        return range(1, 2)  # Only the value [1]
    elif n % 2 == 0:
        return range(2, n+1, 2)
    else:
        return range(1, n+1, 2)
```

函数 alt_range()的结果可以是偶数也可以是奇数。如果 n 是奇数，则结果只包含到 n(含) 的奇数值；如果 n 是偶数，则结果只包含到 n 的偶数值。这种序列对于实现半阶乘函数或双阶乘函数 $n!!$ 尤为重要。

下面介绍如何将 prod()函数和 alt_range()函数组合成一个新的柯里化函数。

```
>>> semi_fact = prod * alt_range
>>> semi_fact(9)
945
```

这里 PyMonad 的*运算符将两个函数组合成了复合函数 semi_fact。它对参数使用函数 alt_range()，随后 prod()函数会使用 alt_range 函数的结果。

使用 PyMonad 的*运算符等同于创建一个新的匿名函数对象。

```
semi_fact = lambda x: prod(alt_range(x))`
```

与创建一个新的匿名函数对象相比，柯里化函数的复合涉及的语法要少一些。

理想情况下，我们希望能如下所示使用函数式复合和柯里化函数：

```
sumwhile = sum * takewhile(lambda x: x > 1E-7)
```

这样就可以定义一个适用于无穷序列的 sum()函数版本，并且在满足阈值条件时停止生成新的值。然而由于 PyMonad 库处理无穷迭代对象的能力似乎不及处理内部 List 对象的能力，因此这种做法实际上不起作用。

14.4　函子和应用型函子

函子指的是简单数据的函数式表示。数字 3.14 的函子版本是一个返回该值的零参数函数。示例如下：

```
>>> pi = lambda: 3.14
>>> pi()
3.14
```

这样就创建了一个零参数匿名函数对象并返回了一个简单的值。

当对函子应用柯里化函数时，会创建一个新的柯里化函子。可以将这一概念概括为：通过使用函数来表示参数、值和函数本身，可以将函数应用于参数来获取值。

一旦程序中的所有内容都是函数，那么所有运算都只是函数式复合模型的一个变体。柯里化函数的参数和结果都可以是函子。有时可以对 `functor` 对象使用 `getValue()`方法，以获得可用于非柯里化代码且适用于 Python 的简单类型。

由于此时编程是基于函数式复合的，因此在实际使用 `getValue()`方法请求一个值之前，无须进行任何计算。程序不会执行大量中间计算，而会定义复杂的中间对象以根据请求生成所需的值。原则上，更智能的编译器或运行时系统能优化这种复合。

当把函数应用于 `functor` 对象时，我们会使用一个类似于 `map()`的方法，相当于*运算符。可以通过 `function * functor` 或 `map(function, functor)`方法理解函子在表达式中的作用。

为了恰当地处理具有多个参数的函数，可以使用`&`运算符构建复合函子。`functor & functor` 方法常用于从一对函子中构建出一个 `functor` 对象。

可以使用 `Maybe` 函子的子类封装 Python 的简单类型。函子 `Maybe` 的有趣之处在于可以用它恰当地处理缺失数据。第 11 章采用的方法是将内置函数装饰为可感知 `None` 值的函数。PyMonad 库采用的方法是装饰数据，以将其排除在外。

函子 `Maybe` 有以下两个子类：

- `Nothing`
- `Just(某个简单值)`

使用 `Mothing` 来代替 Python 中的简单值 `None`，以此表示缺失数据。使用 `Just(某个简单值)`来封装其他所有 Python 对象。这些函子是常数值的类函数表达形式。

可以对这些 `Maybe` 对象使用柯里化函数来恰当地处理缺失数据，示例如下：

```
>>> x1 = systolic_bp * Just(25) & Just(50) & Just(1) & Just(0)
>>> x1.getValue()
116.09

>>> x2 = systolic_bp * Just(25) & Just(50) & Just(1) & Nothing
>>> x2.getValue() is None
True
```

运算符*复合了带参数复合的 `systolic_bp()`函数。运算符&构建了一个复合函子，可以将

该函子作为参数传递给多参数的柯里化函数。

这表明得到的是一个值，而不是一个 TypeError 异常。在处理可能存在缺失数据或无效数据的大型复杂数据集时，这样处理会很方便，比把所有函数都装饰为可感知 None 值要好得多。

这种做法对柯里化函数非常有效。由于函子的方法非常少，因此不能在非柯里化 Python 代码中处理 Maybe 函子。



对于非柯里化 Python 代码，必须使用 getValue()方法来获取简单的 Python 值。

使用惰性 List()函子

刚接触函子 List()的人可能会感到困惑，不同于 Python 内置的 list 类型，它是非常“懒惰的”。当对内置的 list(range(10))方法求值时，list()函数会对 range()对象求值来创建一个包含 10 个项的列表。然而，PyMonad 的 List()函子“懒惰”到甚至不会进行这种计算。

下面比较两者：

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> List(range(10))
[range(0, 10)]
```

函子 List()没有对 range()对象进行求值，它只是在未求值的情况下先保留了该对象。对于收集但不求解函数的情况，函数 pymonad.List()十分有用。

其中 range()的使用可能会令人困惑。Python 3 中的 range()对象同样也是惰性的，因此示例中便包含两层延后。pymonad.List()会根据需求创建数据项。List 中的每一项都是一个可被求值并生成一个值序列的 range()对象。

可以根据之后的需求对 List 函子进行求值：

```
>>> x = List(range(10))
>>> x
[range(0, 10)]
>>> list(x[0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

这样就创建了一个包含 range()对象的惰性 List 对象。随后提取并求解了列表中 0 位置的一个 range()对象。

List 对象不会对生成器函数或 range()对象进行求值，它将任何可迭代参数都视为单个可迭代对象。然而，我们可以使用*运算符来扩展生成器或 range()对象的值。



请注意，*运算符有几种含义：它是内置的数学乘法运算符、PyMonad 定义的函数式复合运算符，以及在调用函数时将单个序列对象绑定为函数所有位置参数的内置修饰符。下面将使用它的第三种含义将一个序列赋给多个位置参数。

range()函数的柯里化版本如下，其下界是 1 而不是 0。对于某些数学运算，这样做会比较方便，因为可以避免内置 range()函数中位置参数的复杂性。

```
@curry
def range1n(n):
    if n == 0: return range(1, 2)  # Only the value 1
    return range(1, n+1)
```

这样就简单地封装了内置的 range()函数，并通过 PyMonad 包将其柯里化了。

由于 List 对象是一个函子，因此可以将函数映射到 List 对象上。

将函数应用于 List 对象中的每一项，示例如下：

```
>>> fact= prod * range1n
>>> seq1 = List(*range(20))
>>> f1 = fact * seq1
>>> f1[:10]
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

这样就定义了一个复合函数 fact()，它由前面的 prod()函数和 range1n()函数构建而来，它是一个阶乘函数；并且创建了一个 List()函子 seq1，它是一个包含 20 个值的序列；还将 fact()函数映射至了 seq1 函子，从而创建了一个阶乘值的序列 f1；还查看了前 10 个值。



多个函数的复合与函数和函子的复合之间存在一定的相似性。prod*range1n 和 fact*seq1 都使用函数式复合，显然前者复合的都是函数，而后者复合的是函数和函子。

用于扩展该示例的另一个小函数如下：

```
@curry
def n21(n):
    return 2*n+1
```

该 n21()小函数执行了一个小型计算。然而由于它是柯里化的，因此可以将其应用于一个函子，如 List()函数。上述示例的第二部分如下所示：

14

```
>>> semi_fact= prod * alt_range
>>> f2 = semi_fact * n21 * seq1
>>> f2[:10]
[1, 3, 15, 105, 945, 10395, 135135, 2027025, 34459425, 654729075]
```

这样就通过此前的 prod()函数和 alt_range()函数定义了一个复合函数。函数 f2 是一个半阶乘（双阶乘）函数。通过将小函数 n21()映射到 seq1 序列来构建函数 f2 的值，便创建了

一个新的序列。随后将 semi_fact 函数应用于这个新序列，创建出了与原序列值对应的一组序列值。

接下来可以将/运算符映射至 map()和 operator.truediv 并行算子了。

```
>>> 2*sum(map(operator.truediv, f1, f2))
3.1415919276751456
```

函数 map()会将给定的运算符应用于两个算子，并生成一个可新增分数的序列。



方法 f1 & f2 会基于两个 List 对象创建出值的所有组合。这是 List 对象的重要特性之一——易于枚举所有组合,这使得通过简单的算法便能计算和过滤出所有可行的备选子集。但这并不是我们想要的，这就是使用 map()函数而不是 operator.truediv * f1 & f2 方法的原因。

我们利用一些函数式复合技术和一个函子的类定义，定义了一个相当复杂的计算。该计算的完整定义如下：

$$\pi = \sum_{n=0}^{\infty} \frac{n!}{(2n+1)!!}$$

理想情况下，我们不希望使用固定大小的 List 对象，而希望有一个惰性求值的、潜在无穷整型数值序列。之后可以使用 sum()函数和 takewhile()函数的柯里化版本计算序列值的总和，直到这些值小到不会对结果产生实在影响。这就需要一个比 List()对象惰性更强的版本，以配合 itertools.counter()函数使用。PyMonad 1.3 中没有这样的潜在无穷列表，因此只能使用固定大小的 List()对象。

14.5 单子的 bind()函数和>>运算符

PyMonad 库的名称来自单子的函数式编程概念，即一个函数按严格的顺序求值。许多函数式编程背后都有一个基本假设：求值顺序是任意的，可以按需优化或重新排列。单子则是例外，它强制从左到右严格按顺序求值。

如前所述，Python 的执行顺序是严格的，因此它不**需要**单子，但我们仍然可以应用此概念将复杂算法清晰化。

强制严格按顺序求值的技术是单子和返回单子的函数之间的一个纽带。一个**扁平的**表达式会变成优化编译器无法重新排序的嵌套绑定。函数 bind()映射为>>运算符，因此可以编写如下表达式：

```
Just(some file) >> read header >> read next >> read next
```

上述表达式会转换为以下形式：

```
bind(
    bind(
        bind(Just(some file), read header),
        read next),
    read next)
```

这些 bind()函数确保了表达式从左到右严格按顺序求值。此外，上述表达式是函数式复合的一个示例。当使用>>运算符创建单子时，创建的是一个复杂对象，它会在最终使用 getValue()方法时被求值。

子类 Just()用于创建一个兼容单子的简单对象，该对象封装了一个简单的 Python 对象。

对于高度优化且语法宽松的语言来说，通过单子的概念表达严格求值顺序至关重要。Python 不需要单子是因为它遵循从左到右的严格求值顺序。由于单子并没有为 Python 环境带来一些全新的东西，因此很难展示其特性。的确，单子声明 Python 所遵循的典型严格求值规则的方式有点冗长。

在 Haskell 等语言中，单子对于文件读写等有严格顺序要求的情况来说至关重要。Python 的命令式模式非常类似于 Haskell 的 do 语句块，它有一个隐式的 Haskell>>=运算符用于强制语句的执行顺序。（类似于 Haskell 的>>=运算，Python 使用的是 bind()函数和>>运算符。）

14.6 模拟实现单子

我们期望通过一种**管道**传递单子，即把单子作为参数传递给一个函数，并将类似的单子作为函数的值返回。必须将该函数设计成可以接收并返回类似的数据结构。

下面介绍用于模拟该过程的一个简单的管道，这种模拟可以是正规蒙特卡洛模拟的一部分。我们将从字面上理解蒙特卡洛模拟，并模拟一个掷骰子游戏，这个极其复杂的模拟过程涉及一些有状态的规则。

其中还涉及一些游戏用语。该游戏包含一个掷骰子的人（一名投手）和其他玩家。游戏分为以下两个阶段。

- 第一次掷骰子称为**出骰子**，其中包含 3 个条件。
  - 如果点数是 7 或 11，则投手赢得游戏。任何投注**过**线的人都会作为胜者赢得奖励，其他人则失去押注。游戏结束，投手可以开始新一轮游戏。
  - 如果点数是 2、3 或 12，则投手输掉游戏。任何投注**不过**线的人赢得奖励，其他人则失去押注。游戏结束，投手必须将骰子转交给另一名投手。
  - 任何其他点数（即 4、5、6、8、9 或 10）会建立一个**点**。游戏状态从**出骰子**变为**点骰子**，游戏继续。

- 一旦建立了点，每一次的**点骰子**都会通过以下 3 个条件来验证。
  - 如果点数是 7，则投手输掉游戏。除了投注**不过**线的人和竟猜注码，其他人全都输掉游戏，游戏结束。由于投手输掉了游戏，因此骰子转交给其他投手。
  - 如果点数和原来一样，则投手输掉游戏。任何投注**过**线的人成为胜者赢得奖励，其他人输掉游戏。游戏结束，投手可以开始新一轮游戏。
  - 任何其他点数的情况，游戏继续进行。一些竟猜注码可能在中间投掷过程中获利或失利。

可以将这些规则视为对状态改变的需求，或者将其视为一系列操作而不是状态改变。必须先使用一个函数再使用递归函数。通过这种方式，配对函数的设计十分适合单子设计模式。

在实际操作中，游戏期间可以投注许多复杂的竟猜注码。可以将它们从游戏的基本规则中分离出来另行求值。其中许多投注（竟猜注码、现场押注等）都是在游戏的**点骰子**阶段进行的。我们将忽略这些额外的复杂性，而只关注核心游戏。

需要一组源随机数，如下所示：

```
import random
def rng():
    return (random.randint(1,6), random.randint(1,6))
```

上述函数生成一组骰子对。

整个游戏的输出结果应如下所示：

```
def craps():
    outcome = (
        Just(("", 0, [])) >> come_out_roll(dice)
                          >> point_roll(dice)
    )
    print(outcome.getValue())
```

这样就创建了一个初始单子 `Just(("",0, []))`，用于定义要使用的基本类型。游戏将生成一个包含结果文本、骰子点值和投掷序列的三元组。每次游戏开始，默认的三元组会构建三元组类型。

把这个单子传递给其他两个函数，这样会根据游戏结果创建出一个结果单子 `outcome`。按照函数必须遵循的执行顺序使用`>>`运算符，将它们以特定顺序相连接。在具有优化编译器的语言中，这会避免表达式顺序重排。

最后使用 `getValue()`方法获取单子的值。由于单子对象是惰性求值的，因此调用该方法会触发对各个单子的求值，以此创建所需的输出。

函数 `come_out_roll()`以柯里化函数 `rng()`为第一个参数，以单子为第二个参数。函数

come_out_roll()会掷骰子并通过**出骰子**规则来确定结果是赢是输还是一个点。

函数 point_roll()也以柯里化函数 rng()为第一个参数，以单子为第二参数。随后函数 point_roll()会掷骰子以确定游戏是否结束。如果尚未结束，则该函数会进行递归运算以找寻最终解。

函数 come_out_roll()如下所示：

```
@curry
def come_out_roll(dice, status):
    d = dice()
    if sum(d) in (7, 11):
        return Just(("win", sum(d), [d]))
    elif sum(d) in (2, 3, 12):
        return Just(("lose", sum(d), [d]))
    else:
        return Just(("point", sum(d), [d]))
```

掷一次骰子来确定第一次的投掷是赢是输还是建立点数，返回的是一个合适的单子，其中包括结果、点值和骰子的投掷。点值对于中间的赢或输没有太大意义。由于没有建立任何点值，因此这里合理地返回一个 0 值。

函数 point_roll()如下所示：

```
@curry
def point_roll(dice, status):
    prev, point, so_far = status
    if prev != "point":
        return Just(status)
    d = dice()
    if sum(d) == 7:
        return Just(("craps", point, so_far+[d]))
    elif sum(d) == point:
        return Just(("win", point, so_far+[d]))
    else:
        return (
            Just(("point", point, so_far+[d]))
            >> point_roll(dice)
        )
```

上面将 status 单子分解为元组的 3 个独立值。既可以使用小型匿名函数对象获取第一个、第二个和第三个值，也可以使用 operator.itemgetter()函数获取元组中的项，但这里使用的是多重赋值语句。

如果尚未建立点，则它的前一个状态是**赢**或者**输**。游戏在一次投掷后便结束了，而且这个函数会仅返回 status 单子。

如果已经建立了一个点，那么其状态便是一个**点**。当掷出骰子后，规则便适用于新一轮投掷。

如果投掷结果是 7，则输掉该游戏并返回最终的单子；如果投掷结果是一个点，则赢得该局游戏并返回相应的单子，否则会将略微修改过的单子传递给 point_roll()函数。修改后的 status 单子会将此次投掷记录在历史投掷中。

典型的输出结果如下所示：

```
>>> craps()
('craps', 5, [(2, 3), (1, 3), (1, 5), (1, 6)])
```

最终的单子有一个用于显示结果的字符串，其中包含建立的点值和骰子的投掷序列。每个结果都对应一个特定奖励（payout），可以根据它来确定投注者所持押注的总体波动。

可以通过模拟来检验不同的投注策略。我们希望寻找一种方法来破除游戏中任何隐含的主场优势。



游戏的基本规则中存在有一些细小的不对称性。掷得 11 直接赢和掷得 3 直接输是相互平衡的。主场有 5.5%（1/18≈0.055）的优势是因为掷得 2 和 12 也算输。这里要考虑哪些额外的投注机会可以削弱这一主场优势。

利用一些简单的函数式设计技术可以构建大量智能的蒙特卡洛模拟。特别是当存在复杂顺序或内部状态时，单子有助于构建这类计算。

14.7　单子的其他特性

PyMonad 的另一个特点是名字容易混淆的**幺半群**（monoid）。它源自数学，指的是具有运算符和标识元素的一组数据元素，且这组数据元素相对于运算符是封闭的。自然数、add 运算符和标识元素 0 可以组成幺半群。正整数、*运算符和标志元素 1 也能构成幺半群。使用|作为运算符且使用空字符串作为标识元素也能构成幺半群。

PyMonad 包含许多预定义的幺半群类。可以扩展它们来添加自己的 monoid 类，以限制编译器只进行某些优化。还可以使用幺半群类来创建可以累积复杂值的数据结构，其值可能包括过往操作的历史记录。

本书大部分内容都有助于读者深入理解函数式编程。理解文档是学习函数式编程的一条捷径。与其学习整门语言和整套工具集来编译并运行函数式程序，不如直接通过交互式 Python 进行试验。

实际上，并不需要太多特性，因为 Python 已经是有状态的，并规定了严格的表达式求值顺序。没有必要在 Python 中引入状态化对象或严格有序求值。结合函数式概念和 Python 的命令式实现，可以用 Python 编写出有用的程序。出于这个原因，本书不再深入探讨 PyMonad。

14.8 小结

本章介绍了如何在 Python 中使用 PyMonad 库来展现函数式编程的一些概念。该模块包含了许多重要的函数式编程技术。

然后介绍了柯里化概念，该函数通过组合参数来生成新函数。柯里化函数使得我们可以使用函数式复合，根据简单的函数创建出复杂的函数。还介绍了可以将简单数据对象封装为函数的函子，它也可以用于函数式复合。

在使用优化编译器和惰性求值规则时，可以用单子指定严格的求值顺序。Python 中并没有很好的单子用例，因为 Python 在底层是一种命令式编程语言。在某些情况下，命令式 Python 可能比构造单子更简洁明了。

下一章将介绍如何运用函数式编程技术构建 Web 服务和应用。HTTP 的思想可以概括为 `response = httpd(request)`。理想情况下，HTTP 是无状态的，因而非常适合函数式编程。cookie 的使用类似于为之后的请求提供响应值作为参数。

第 15 章

Web 服务的函数式设计方法

现在搁置 EDA 主题，转向 Web 服务器和 Web 服务。Web 服务器在某种程度上就是许多函数的级联。可以应用许多函数式设计模式来解决 Web 内容的呈现问题。本章会介绍如何使用 REST（representational state transfer，表述性状态转移），以及使用函数式设计模式构建基于 REST 的（RESTful）Web 服务。

我们无须发明新的 Python Web 框架，也不想从可用框架中选取一个。Python 中有许多可用的 Web 框架，并且每个框架都有各自的特性和优点。

本章旨在介绍一些适用于大多数可用框架的设计原则，它们有助于我们利用函数式设计模式呈现 Web 内容。

在研究极其庞大或复杂的数据集时，可能需要支持子集化或搜索的 Web 服务，以及可以下载到各种格式的数据子集的网站。在这种情况下，可能需要使用函数式设计创建基于 REST 的 Web 服务，来满足这些复杂的需求。

交互式 Web 应用程序通常依赖有状态会话来使站点更易用。用户的会话信息由 HTML 表单提供的数据、从数据库获取的数据，或者从之前交互的缓存中回收的数据等来更新。由于有状态数据必须作为每个事务的一部分来获取，因此它更像是输入参数或者返回值。即便在存在 cookie 和数据库更新的情况下，这样也可以实现函数式编程。

本章将讨论以下几个主题：

- HTTP 请求和响应模型的基本概念；
- Python 应用程序使用的 WSGI（Web 服务网关接口）标准；
- 使用 WSGI 将 Web 服务定义为函数，这符合无状态服务器的 HTTP 理念；
- 以及如何授权客户端应用程序以接入 Web 服务。

15.1 HTTP“请求－响应”模型

HTTP 协议几乎是无状态的，即用户代理（或浏览器）发出请求后服务器提供响应。对于不

涉及 cookie 的服务，客户端程序可以从函数式角度处理协议。我们可以使用 http.client 和 urllib 库来构建客户端。一个 HTTP 用户代理实际执行的内容如下所示：

```
import urllib.request
def urllib_demo(url):
    with urllib.request.urlopen(url) as response:
        print(response.read())

urllib_demo("http://slott-softwarearchitect.blogspot.com")
```

wget 和 curl1 等程序使用以命令行参数形式提供的 URL 来完成这种处理。浏览器会响应用户的指示和点击操作，URL 则来自用户的操作，这种操作通常是指点击链接文本或图像。

然而，对 UX（user experience，用户体验）设计的实际考量会导致一些有状态的实现细节。当客户端必须跟踪 cookie 信息时，它就会变成有状态的。响应头会提供 cookie，并且后续的请求必须将 cookie 返回给服务器，稍后将详细讨论该问题。

HTTP 响应会包含一个需要用户代理端进行额外操作的状态码。300~399 范围内的许多状态码表明请求的资源已经被移走了。用户代理随后需要保存来自 Location 报头的详细信息，并请求一个新的 URL。401 状态码表明需要认证，因此用户代理必须使用包含访问服务器所需凭证的 Authorization 报头来发起新的请求。库 urllib 可以处理有状态客户端的请求。库 http.client 不会自动跟随 3xx 的重定向状态码。

用户代理处理 3xx 和 401 代码的技术可以通过简单的递归实现。如果状态不表示重定向，则为基础情况，函数返回结果。如果需要重定向，则可以使用重定向地址递归地调用该函数。

考虑协议的另一面：一个静态内容的服务器可以是无状态的。对于此，可以如下所示使用 http.server 库：

```
from http.server import HTTPServer, def server_demo():
httpd = HTTPServer(
    ('localhost', 8080), SimpleHTTPRequestHandler)
while True:
    httpd.handle_request()
httpd.shutdown()
```

上面创建了一个服务器对象，并将其赋给了 httpd 变量。我们提供了监听连接请求所需的地址和端口号。TCP/IP 协议会在一个单独的端口创建新的连接。HTTP 协议会从这个端口读取请求并创建该句柄的实例。

在本例中，我们提供了 SimpleHTTPRequestHandler 类来初始化每个请求。这个类必须实现一个最小的接口子集，用于向客户端先发送报头再发送响应体。这个特定类会提供本地目录中的文件。如果想自定义功能，可以创建一个子类，实现 do_GET() 和 do_GET() 等方法来改变其行为。

通常使用 `serve_forever()` 方法，而不是手动编写循环。这里展示循环的使用是为了说明如果需要停止服务器，则必须关闭服务器。

15.1.1　通过 cookie 注入状态

Cookie 的引入改变了客户端和服务器之间的整体关系，使之状态化了，然而没有涉及对 HTTP 协议本身的修改。状态信息通过请求和响应的报头进行通信。服务器会在响应报头中向用户代理发送 cookie。用户代理会在请求报头中保存并使用 cookie 进行响应。

用户代理或浏览器需要保留 cookie 值的缓存，将它作为响应的一部分来提供，并在后续的请求中包含相应的 cookie。Web 服务器会在请求报头中查找 cookie，并在响应报头中提供更新了的 cookie。这样做的效果是 Web 服务器变为无状态的，而状态的改变只在客户端发生。服务器将 cookie 视为请求中的额外参数，并会在响应中提供一些附加的细节信息。

Cookie 可以包含任何内容。通常会将它们加密以避免 Web 服务器的详细信息暴露给客户端计算机上运行的其他应用程序。传输巨大的 cookie 会拖慢处理速度。优化这种状态处理的最佳方法是使用现有框架。我们将忽略 cookie 和会话管理的细节。

会话的概念是 Web 服务器的一个特性，而不是 HTTP 协议。通常将会话定义为具有相同 cookie 的一系列请求。在发出初始请求时，由于没有可用的 cookie，因此会创建一个新的会话 cookie。每个后续请求都将包含该 cookie。一个登陆用户的会话 cookie 中会包含其他细节信息。只要服务器还继续接收该 cookie，会话便可以一直存续，即 cookie 可以永久有效，或者在几分钟后失效。

Web 服务的 REST 方法不依赖会话或 cookie。由于每个 REST 请求都不同，因此相比使用 cookie 来简化用户交互的交互式网站，它不是那么**用户友好**。本书关注基于 REST 的 Web 服务是因为它们非常符合函数式设计模式。

使用无会话 REST 进程的一个结果是每个单独的 REST 请求都会被单独验证。如果需要身份认证，则意味着 REST 通信必须使用 SSL（secured socket layer，安全套接字层）协议。可以使用 `https` 方案将凭证从客户端安全地传输到服务器。

15.1.2　函数式设计的服务器考量

HTTP 背后的一个核心思想是服务器响应是对请求的函数。从概念上讲，Web 服务应该具有顶层实现，概括如下：

```
response = httpd(request)
```

然而这样做是不切实际的。事实证明，HTTP 请求并不是简单的、单一的数据结构，它包含一些必要的内容和可选的内容。请求可能含有报头、方法和路径，也可能有附件。这里的附件可

能包括表单或上传的文件，或两者兼有。

如果情况再复杂一些，那么可以将浏览器的表单数据作为 `GET` 请求路径中的查询字符串来发送，或者将其作为附件发送给 `POST` 请求。尽管可能会造成混淆，但大部分 Web 应用程序框架都会创建 HTML 表单标签，通过`<form>`标签中的 `method=POST` 参数提供数据，随后表单数据会以附件形式包含在请求中。

15.1.3 深入研究函数式视图

HTTP 响应和请求都有独立于主体的报头。请求还可以包含一些附加的表单数据或者其他上传的文件，因此可以把一个 Web 服务器视为如下形式：

```
headers, content = httpd(headers, request, [form or uploads])
```

请求报头可能包含 cookie 值，可以将其视为额外添加的参数。此外，Web 服务器通常依赖所运行的操作系统环境，因此也可以将该操作系统环境数据看作请求中提供的附加参数。

对于内容，有一个范围虽广但十分合理的定义。MIME（multipurpose internet mail extensions，多用途互联网邮件扩展）类型定义了 Web 服务可能返回的内容类型。其中包括纯文本、HTML、JSON、XML，或者网站提供的非文本形式的各种媒体。

当深入研究针对 HTTP 请求构建响应所需的处理时，会发现一些我们希望复用的公共特性。可复用元素的思想是创建从简单到复杂的 Web 服务框架。函数式设计的方式允许我们复用函数，这表明函数式方法有助于构建 Web 服务。

下面介绍如何为服务响应中的各个元素创建管道，来讲解 Web 服务的函数式设计。我们将通过嵌套负责处理请求的函数来实现这一点，这样外部元素产生的一般开销就不会影响内部元素了。这也使得可以把外部元素作为过滤器，为无效的请求生成错误响应，并让内部函数只关注应用程序本身的处理。

15.1.4 嵌套服务

可以将 Web 请求处理看作许多嵌套的上下文。例如外层上下文可能涉及会话管理，即检查请求以确定这是现有会话还是新会话中的一个请求；内层上下文可能会为用于检测 CSRF（Cross-Site Request Forgeries，跨站请求伪造）的表单处理提供令牌；其他上下文可能会处理会话中的用户身份认证。

对于上面解释的功能，其概念性的视图如下所示：

```
response = content(
    authentication(
        csrf(
```

```
                session(headers, request, forms)
            )
        )
    )
```

这里的思想是每个函数都可以建立在前一个函数的结果之上。每个函数既可以扩充请求，也可以因请求无效而将其拒绝。例如函数 `session()`可以使用报头来确定这是一个现有会话还是一个新会话。由于 CSRF 处理要求会话有效，因此函数 `csrf()`会检查表单输入以确保使用的是合适的令牌。对于缺少有效凭证的会话，函数 `authentication()`会返回一个错误响应，而当存在有效凭证时，它可以使用用户信息来扩充请求。

函数 `content()`无须担心会话、伪造和未经身份认证的用户。它可以专注于解析路径，以确定应当提供哪类内容。在更复杂的应用程序中，函数 `content()`可能包含极其复杂的映射，用于将路径元素映射到能确定合适内容的函数。

然而嵌套的函数视图仍然不够准确，其问题在于每个嵌套的上下文可能还需要调整响应，而不是或者不仅是调整请求。

理想的情况如下所示：

```
def session(headers, request, forms):
    pre-process: determine session
    content = csrf(headers, request, forms)
     post-processes the content
     return the content

def csrf(headers, request, forms):
    pre-process: validate csrf tokens
    content = authenticate(headers, request, forms)
    post-processes the content
    return the content
```

这一概念体现出一种函数式设计思想：可以使用支持扩充输入或输出的嵌套函数集合来创建 Web 内容。更巧妙的做法是，应当定义可供不同函数使用的简单标准接口。一旦将该接口标准化了，就能以不同的方式组合函数并添加功能了，从而实现函数式编程的目标，用简单明了的程序提供 Web 内容。

15.2　WSGI 标准

WSGI（web server gateway interface，Web 服务网关接口）定义了一个相对简单且标准化的设计模式，用于创建对 Web 请求的响应。这是大多数基于 Python 的 Web 服务器的通用框架。更多相关信息，可访问 http://wsgi.readthedocs.org/en/latest/。

关于 WSGI 的一些背景知识，可访问 https://www.python.org/dev/peps/pep-0333。

Python 库中的 wsgiref 包包含了一个 WSGI 的参考实现。每个 WSGI 应用程序都具有相同的接口，如下所示：

```
def some_app(environ, start_response):
    return content
```

参数 environ 是一个字典，以单一且统一的结构包含了请求的所有参数。报头、请求方法、路径以及任何表单和上传文件的附件都会存放在这个环境中。除此之外，伴随 WSGI 请求处理的一些项目还提供了操作系统级的上下文。

参数 start_response 是一个专门用于发送状态和响应报头的函数。WSGI 服务器中最终负责构建响应的部分将使用给定的 start_response() 函数，并构建响应文档作为返回值。

从 WSGI 应用程序返回的响应是字符串序列，或是会返回给用户代理的类字符串文件封装器。如果使用 HTML 模板工具，那么该序列可能只含有一项。在某些情况下（例如 Jinja2 模板）会将模板惰性渲染为文本块序列，这允许服务器交错执行模板填充和用户代理的下载。

可以对 WSGI 应用程序使用如下类型提示：

```
from typing import (
    Dict, Callable, List, Tuple, Iterator, Union, Optional
)
from mypy_extensions import DefaultArg

SR_Func =  Callable[
    [str, List[Tuple[str, str]], DefaultArg(Tuple)], None]

def static_app(
    environ: Dict,
    start_response: SR_Func
) -> Union[Iterator[bytes], List[bytes]]:
```

类型定义 SR_Func 是函数 start_response 的签名。请注意，该函数有一个可选参数，要求用 mypy_extensions 模块中的函数来定义该特征。

static_app() 作为整个 WSGI 函数，需要使用环境参数和 start_response() 函数。返回的结果是字节序列，或者是基于字节的迭代器。可以扩展函数 static_app() 返回类型的组合，使其包含 BinaryIO 和 List[BinaryIO]，但本章的任何示例都不会用到它们。

截至本书出版时，wsgiref 包中尚没有完整的类型定义集合。具体而言，wsgiref.simple_server 模块缺少合适的存根定义（stub definition），因此 mypy 会发出警告。

每个 WSGI 应用程序都设计成了函数的集合。可以将该集合视为嵌套的函数或一条转换链。链中的每个应用程序都会返回一个错误，或者将请求转交给另一个应用程序来确定最终结果。

通常用 URL 路径来确定要使用的备选应用程序，这会形成一个 WSGI 应用程序的树形结构，

并且它们可以共享公共组件。

下面是一个非常简单的路由应用程序，它获取 URL 路径中的第一个元素，并以此定位另一个提供内容的 WSGI 应用程序。

```
SCRIPT_MAP = {
    "demo": demo_app,
    "static": static_app,
    "index.html": welcome_app,
}
def routing(environ, start_response):
    top_level = wsgiref.util.shift_path_info(environ)
    app = SCRIPT_MAP.get(top_level, welcome_app)
    content = app(environ, start_response)
    return content
```

该应用程序会使用 wsgiref.util.shift_path_info()函数来调整环境。它会对 environ['PATH_INFO']字典中请求路径中的各项进行**头尾分割**。第一个/之前的路径头部会移至环境中的 SCRIPT_NAME 项，PATH_INFO 项则由路径的尾部内容进行更新。返回的值也会作为路径头部，其值和 environ['SCRIPT_NAME']一致。在没有路径可解析的情况下，返回值为 None，且不对环境进行更新。

函数 routing()使用路径中的第一项来定位 SCRIPT_MAP 字典中的应用程序。我们使用 welcome_app 作为默认设置，避免请求的路径不符合映射规则，这似乎比返回 HTTP 的 404 NOT FOUND 错误要好一些。

该 WSGI 应用程序是一个函数，用于选取其他 WSGI 函数。请注意，路由函数并不返回函数，而是为所选取的 WSGI 应用程序提供修改过的环境。这是将任务从一个函数转移到另一个函数的典型设计模式。

框架通过正则表达式来泛化路径匹配过程。可以设想使用**正则表达式**序列和 WSGI 应用程序来配置 routing()函数，而不是使用字符串到 WSGI 应用程序的映射。改进的 routing()函数应用程序会计算每个正则表达式来寻找匹配项。在找到匹配后，在调用所请求的应用程序之前可以使用任何 match.groups()函数来更新环境。

15.2.1 在 WSGI 处理期间抛出异常

WSGI 应用程序的一个核心特性是链上的每个阶段都会负责过滤请求。其思想是在处理过程中尽早拒绝错误的请求。Python 的异常处理使这一点易于实现。

可以定义一个提供静态内容的 WSGI 应用程序，如下所示：

```
def static_app(
        environ: Dict,
```

```
    start_response: SR_Func
) -> Union[Iterator[bytes], List[bytes]]:
log = environ['wsgi.errors']
try:
    print(f"CWD={Path.cwd()}", file=log)
    static_path = Path.cwd()/environ['PATH_INFO'][1:]
    with static_path.open() as static_file:
        content = static_file.read().encode("utf-8")
        headers = [
            ("Content-Type", 'text/plain;charset="utf-8"'),
            ("Content-Length", str(len(content))),
        ]
        start_response('200 OK', headers)
        return [content]
except IsADirectoryError as e:
    return index_app(environ, start_response)
except FileNotFoundError as e:
start_response('404 NOT FOUND', [])
return [
    f"Not Found {static_path}\n{e!r}".encode("utf-8")
]
```

该应用程序从当前工作目录中创建了一个 `Path` 对象，以及一个作为请求 URL 一部分而提供的路径元素。路径信息是 WSGI 环境的一部分，位于以 `PATH_INFO'` 为键的项中。正是由于这种解析路径的方式，因此它会有一个前导的`/`，我们使用 `environ['PATH_INFO][1:]`来丢弃该字符。

该应用程序尝试将请求的路径以文本文件的形式打开。这里有两个常见问题，会将它们作为异常来处理：

- 如果文件是一个目录，将使用另一个应用程序 `index_app` 来呈现目录内容；
- 如果没有找到该文件，将返回一个 `HTTP 404 NOT FOUND` 响应。

该 WSGI 应用程序引发的其他任何异常都不会被捕获。应把调用该应用程序的程序设计为具有某种通用的错误响应能力。如果应用程序不处理这些异常，则会使用通用的 WSGI 故障响应。



上面的处理涉及严格的运算顺序。必须读取整个文件才能创建一个正确的 HTTP `Content-Length` 报头。

处理异常有两种方式。一种是调用另一个应用程序。如果需要向该应用程序提供额外信息，则必须使用所需的信息来更新环境。这便是为一个复杂网站构建标准化错误页面的方法。

另一种是调用 `start_response()`函数并返回一个错误结果，这适用于特定的本地化行为。最终的内容以字节形式呈现，这意味着必须将 Python 字符串正确编码，且必须向用户代理提供编码信息，甚至在错误信息 `repr(e)`被下载之前就应该已经正确编码。

15.2.2　实用的 WSGI 应用程序

设计 WSGI 标准的目标不是定义一个完整的 Web 框架，而是定义一组最小标准集合来支持灵活的、Web 处理相关的互操作性。一个框架可以采用完全不同的方法来提供 Web 服务。最外层的接口应该与 WSGI 标准相兼容，以便在不同的上下文中使用。

Apache 的 httpd 和 Nginx 等 Web 服务器都有适配器，用于从 Web 服务器向 Python 应用程序提供兼容 WSGI 的接口。有关 WSGI 实现的更多信息，可参考 https://wiki.python.org/moin/WSGIImplementations。

将应用程序嵌入一个更大的服务器中可以清晰分离我们所关注的问题。可以使用 Apache httpd 提供完全静态的内容，如`.css`、`.js` 和图像文件等。而对于 HTML 页面，NGINX 等服务器可以使用 `uwsgi` 模块将请求传递给单独的 Python 进程，并由该进程负责处理 Web 内容中我们感兴趣的 HTML 部分。

分离静态内容和动态内容意味着必须创建一个单独的媒体服务器，或者为网站定义两组路径。如果采取第二种方法，那么某些路径将包含完全静态的内容，并且可交由 Nginx 处理。其他包含动态内容的路径则交由 Python 处理。

在处理 WSGI 函数时，需要注意不能修改或扩展 WSGI 接口，这保证了应用程序外部可见层的完全兼容。内部的结构和处理不必符合 WSGI 标准，外部接口必须统一遵循这些规则。

WSGI 定义的一个结果是常用额外的配置参数更新 `environ` 字典。通过这种方式，一些 WSGI 应用程序可以充当网关，利用从 cookie、配置文件或数据库中提取的信息来扩充环境。

15.3　将 Web 服务定义为函数

下面介绍一个基于 REST 的 Web 服务，它可以切分源数据，并提供 JSON、XML 或 CSV 格式文件的下载。我们将提供一个完全兼容 WSGI 的封装器。负责执行应用程序实际工作的函数不必严格遵循 WSGI 标准。

我们会使用一个具有 4 个子集的简单数据集：安斯库姆四重奏。第 3 章介绍过读取和解析这些数据的方法。它是一个小型数据集，但可以用来展示基于 REST 的 Web 服务的基本原理。

该应用程序将分为两层：属于简单 WSGI 应用程序的 Web 层，和使用了更典型的函数式编程技术的数据服务层。首先介绍 Web 层，这样之后就可以专注于提供有意义结果的函数式方法了。

需要为该 Web 服务提供以下两种信息。

- 所需的四重奏：这是一个切片操作，其思想是通过过滤和提取有意义的子集来分割信息。
- 所需的输出格式。

通常通过请求路径来选取数据。可以请求/anscombe/I/或/anscombe/II/通过四重奏选取特定数据集。其思想是用 URL 来定义资源，并且没有任何理由来改变 URL。在这种情况下，数据集选取器将不依赖日期或某些组织化的批准状态以及其他外部因素。URL 是不包含时间的绝对路径。

输出格式不作为顶层 URL 的一部分，它只是一个序列化格式，而不是数据本身。在某些情况下，格式是通过 HTTP 的 Accept 报头来请求的。这在浏览器中很难使用，但是在使用了 RESTful API 的应用程序中却很容易使用。从浏览器提取数据时，通常通过查询字符串来指定输出格式。我们将在路径末尾使用?form=json 方法来指定 JSON 输出格式。

可以使用如下所示的 URL：

```
http://localhost:8080/anscombe/III/?form=csv
```

这将请求以 CSV 格式下载第 3 个数据集。

15.3.1　创建 WSGI 应用程序

首先使用一个简单的 URL 模式匹配表达式来定义应用程序中的唯一路由。在一个更大或更复杂的应用程序中，模式可能不止一种：

```
import re
path_pat= re.compile(r"^/anscombe/(?P<dataset>.*?)/?$")
```

通过这种模式，可以在路径顶层定义一个关于 WSGI 的完整**脚本**。在本例中，该脚本是 anscombe。我们把下一级路径作为通过安斯库姆四重奏选取的数据集，数据集的值落于 I、II、III 或 IV。

使用命名参数作为选取条件。在许多情况下，可以使用如下语法来描述 RESTful API：

```
/anscombe/{dataset}/
```

这种理想化的模式转化为了适当的正则表达式，并在路径中保留了数据集选择器的名称。

一些 URL 路径示例如下，它们演示了这种模式的工作原理：

```
>>> m1 = path_pat.match( "/anscombe/I" )
>>> m1.groupdict()
{'dataset': 'I'}
>>> m2 = path_pat.match( "/anscombe/II/" )
>>> m2.groupdict()
{'dataset': 'II'}
>>> m3 = path_pat.match( "/anscombe/" )
>>> m3.groupdict()
{'dataset': ''}
```


这些示例都显示了从 URL 路径中解析得到的详细信息。如果指定了某个序列的名称，则它

会出现在路径中；如果没有指定序列名称，则模式返回的是一个空字符串。

完整的 WSGI 应用程序如下：

```
import traceback
import urllib.parse
def anscombe_app(
        environ: Dict, start_response: SR_Func
    ) -> Iterable[bytes]:
    log = environ['wsgi.errors']
    try:
        match = path_pat.match(environ['PATH_INFO'])
        set_id = match.group('dataset').upper()
        query = urllib.parse.parse_qs(environ['QUERY_STRING'])
        print(environ['PATH_INFO'], environ['QUERY_STRING'],
              match.groupdict(), file=log)

        dataset = anscombe_filter(set_id, raw_data())
        content_bytes, mime = serialize(
            query['form'][0], set_id, dataset)
        headers = [
            ('Content-Type', mime),
            ('Content-Length', str(len(content_bytes))),
        ]
        start_response("200 OK", headers)
        return [content_bytes]
    except Exception as e:  # pylint: disable=broad-except
        traceback.print_exc(file=log)
        tb = traceback.format_exc()
        content = error_page.substitute(
            title="Error", message=repr(e), traceback=tb)
        content_bytes = content.encode("utf-8")
        headers = [
            ('Content-Type', "text/html"),
            ('Content-Length', str(len(content_bytes))),
        ]
        start_response("404 NOT FOUND", headers)
        return [content_bytes]
```

该应用程序会从请求中提取两条信息：环境字典中的 `PATH_INFO` 键和 `QUERY_STRING` 键。`PATH_INFO` 请求会定义要提取的数据集。`QUERY_STRING` 请求将指定输出的格式。

请注意，查询字符串可能非常复杂。我们使用了 `urllib.parse` 模块来准确定位查询字符串中所有名称和值的配对，而不是简单地假定它是一个类似于`?form=json` 的字符串。在通过查询字符串提取得到的字典中，可以在 `query['form'][0]`中找到以 `form` 为键的值。这应该是定义过的格式之一，否则会引发异常，并显示一个错误页面。

定位到路径和查询字符串后，用粗体强调了应用程序的处理过程。这两条语句依赖三个函数来对结果进行收集、过滤和序列化。

- 函数 raw_data()从文件中读取原始数据，结果是一个包含一组 Pair 对象的字典。
- 函数 anscombe_filter()接收一个选择字符串和源数据字典，并返回 Pair 对象的单个列表。
- serialize()函数随后将该配对列表序列化为字节码。序列化器会生成字节码，这样就可以用合适的报头打包并返回了。

我们选择了生成一个 HTTP 的 Content-Length 报头作为结果的一部分。这个报头并不是必需的，但有助于下载大文件。由于我们决定发送该报头，因此必须用序列化的数据创建字节对象，以便统计这些字节。

如果选择忽略 Content-Length 报头，那么可以大规模地改变这个应用程序的结构。可以把每个序列化器都更改为生成器函数，并且它们在创建时会生成字节对象。对于大型数据集来说，这种优化可能有益，然而对于关注下载进度的用户可能不太友好，因为浏览器无法显示下载的完成情况。

一种常见的优化是将事务分解为两部分。第一部分用于计算结果并将文件放入 Downloads 目录。响应是一个带有 Location 报头的 302 FOUND，用于标识需下载的文件。通常大部分客户端会基于这个原始响应来请求文件。可以用不涉及 Python 应用的 Apache httpd 或者 Nginx 下载文件。

本例将所有错误视为 404 NOT FOUND 错误会引起误导，因为可能是其他环节出了问题。更复杂的错误处理会引入更多的 try:/except:块来提供更多信息反馈。

出于调试的目的，在生成的 Web 页面中提供了 Python 的栈跟踪。在没有调试需求的环境中，这是一种非常糟糕的做法。来自 API 的反馈应该只用于处理请求，无他。栈跟踪会向潜在的恶意用户提供过多信息。

15.3.2　获取原始数据

函数 raw_data()类似于第 3 章的例子，但有一些重要的变化。该应用程序中用到了如下内容：

```
from Chapter_3.ch03_ex5 import (
    series, head_map_filter, row_iter)
from typing import (
    NamedTuple, Callable, List, Tuple, Iterable, Dict, Any)

RawPairIter = Iterable[Tuple[float, float]]

class Pair(NamedTuple):
    x: float
    y: float

pairs: Callable[[RawPairIter], List[Pair]] \
    = lambda source: list(Pair(*row) for row in source)
```

```
def raw_data() -> Dict[str, List[Pair]]:
    with open("Anscombe.txt") as source:
        data = tuple(head_map_filter(row_iter(source)))
        mapping = {
            id_str: pairs(series(id_num, data))
            for id_num, id_str in enumerate(
                ['I', 'II', 'III', 'IV'])
        }
    return mapping
```

函数 raw_data()会打开本地数据文件，并利用 row_iter()函数将解析后的文件的每一行返回单独的项目行中。上面运用 head_map_filter()函数删除了文件的头部，结果创建了一个列表元组结构，并将其赋给 data 变量，这可以将输入解析为有用的结构。生成的结构是 NamedTuple 的子类 Pair 的一个实例，且其中两个字段的类型提示为 float。

上面使用字典推导式构建了从 id_str 到由 series()函数结果组合配对的映射。函数 series()从输入文档中提取(x, y)对。文档中的每个序列都在两个相邻的列中。名为 I 的序列在第 0 列和第 1 列，函数 series()从中提取相关列对。

创建的函数 pairs()是匿名函数对象，因为它是带有单个参数的小型生成器函数。该函数根据 series()函数创建的匿名元组序列构建所需的 NamedTuple 对象。

由于 raw_data()函数的输出是映射，因此通过名称选取某个特定的序列，示例如下：

```
>>> raw_data()['I']
[Pair(x=10.0, y=8.04), Pair(x=8.0, y=6.95), ...
```

给定一个键，例如 I，该序列是一个 Pair 对象的列表，且该序列中的每一项都具有 x 值和 y 值。

15.3.3　运用过滤器

该应用程序中使用了一个非常简单的过滤器。以下函数展现了整个过滤流程：

```
def anscombe_filter(
        set_id: str, raw_data_map: Dict[str, List[Pair]]
    ) -> List[Pair]:
    return raw_data_map[set_id]
```

把这个简单表达式转化为函数出于以下 3 个原因：

- 相比下标表达式，函数表示法形式上更一致，也更灵活；
- 易于扩展过滤器的功能；
- 可以在这个函数的文档字符串中包含独立的单元测试。

尽管简单的匿名函数可以正常工作，但测试起来并不怎么方便。

我们完全没做任何错误处理，而关注的是所谓的**幸福之路**（happy path），即一系列理想事件。该函数中出现的任何问题都可能引发异常。WSGI 的封装函数应当捕获所有异常并返回相应的状态信息和错误响应内容。

例如 `set_id` 方法有时会出错。与其纠结于所有可能的出错情形，不如直接允许 Python 给出异常。实际上，该函数遵循了 Python 的建议：**寻求原谅而不是请求许可**。通过避免**请求许可**在代码中实践了该建议，即没有任何预设的 `if` 语句会试图判定参数是否有效。这里只有**宽恕**处理——异常出现后即在 WSGI 封装器中进行处理。这一基本建议适用于之前的原始数据和下面要介绍的序列化。

15.3.4　序列化结果

序列化是将 Python 数据转换成适合传输的字节流。每种格式最好用一个简单的函数来描述，该函数只序列化这一种格式。随后可以在特定序列化器列表中选取一个顶层通用序列化器。可使用以下函数集合选择序列化器：

```
Serializer = Callable[[str, List[Pair]], bytes]
SERIALIZERS: Dict[str, Tuple[str, Serializer]]= {
    'xml': ('application/xml', serialize_xml),
    'html': ('text/html', serialize_html),
    'json': ('application/json', serialize_json),
    'csv': ('text/csv', serialize_csv),
}

def serialize(
        format: str, title: str, data: List[Pair]
    ) -> Tuple[bytes, str]:
    mime, function = SERIALIZERS.get(
        format.lower(), ('text/html', serialize_html))
    return function(title, data), mime
```

整个 `serialize()`函数负责在 `SERIALIZERS` 字典中定位某个特定的序列化器，该字典将格式名称映射到一个二元组。该元组有一个 MIME 类型，且必须在响应中使用来表示结果。该元组还包含一个基于 `Serializer` 类型提示的函数。该函数会把名称和 `Pair` 对象列表转换为待下载的字节对象。

函数 `serialize()`没有转换任何数据。它只是将名称映射到负责执行转换的函数，返回一个函数使得整个应用程序可以管理内存的详细情况或文件系统的序列化。尽管对文件系统进行序列化很慢，但它可以处理大的文件。

下面讲解各个序列化器。它们可以分为两组：生成字符串的序列化器和生成字节码的序列化器。生成字符串的序列化器需要将字符串编码为字节以供下载。生成字节码的序列化器无须任何额外操作。

对于生成字符串的序列化器，可以使用带有标准化字节转换函数的复合函数。可以实现标准化字节转换的装饰器如下所示：

```
from typing import Callable, TypeVar, Any, cast

from functools import wraps
def to_bytes(
        function: Callable[..., str]
    ) -> Callable[..., bytes]:
    @wraps(function)
    def decorated(*args, **kw):
        text = function(*args, **kw)
        return text.encode("utf-8")
    return cast(Callable[..., bytes], decorated)
```

这样就创建了一个名为`@to_bytes`的小型装饰器。它会对给定的函数进行求值，然后使用 UTF-8 将结果编码为字节码。请注意，装饰器将被装饰函数的返回值类型从`str`变为了`bytes`。至此，还没有正式声明被装饰函数的参数，以及使用`...`来替代实现细节。下面将展示如何配合 JSON、CSV 和 HTML 序列化器来使用该装饰器。XML 序列化器会直接生成字节码，因此不需要使用这个附加函数进行复合。

也可以在对`serializers`映射的初始化中进行函数式复合。可以装饰函数对象的引用，而非装饰函数定义。序列化器映射的另一种定义如下：

```
SERIALIZERS = {
    'xml': ('application/xml', serialize_xml),
    'html': ('text/html', to_bytes(serialize_html)),
    'json': ('application/json', to_bytes(serialize_json)),
    'csv': ('text/csv', to_bytes(serialize_csv)),
}
```

这样就把函数定义中的装饰器替换为了在构建数据结构映射时使用的装饰器。这种延后装饰的形式可能会令人困惑。

15.3.5　序列化数据为 JSON 或 CSV 格式

JSON 序列化器和 CSV 序列化器很相似，因为它们都依赖 Python 库来进行序列化。这些库本质上都是命令式的，因此函数体由严格的语句序列组成。

JSON 序列化器如下所示：

```
import json

@to_bytes
def serialize_json(series: str, data: List[Pair]) -> str:
    """
    >>> data = [Pair(2,3), Pair(5,7)]
    >>> serialize_json( "test", data )
```

```
    b'[{"x": 2, "y": 3}, {"x": 5, "y": 7}]'
    """
    obj = [dict(x=r.x, y=r.y) for r in data]
    text = json.dumps(obj, sort_keys=True)
    return text
```

这样就创建了一个字典列表的结构，并使用 `json.dumps()`函数创建了字符串表示形式。JSON 模块要求具体化的 `list` 对象，因此不能提供惰性生成器函数。尽管参数值 `sort_keys=True` 有助于单元测试，但它不是应用程序所必需的，而且会引入一些开销。

CSV 序列化器如下所示：

```
import csv
import io

@to_bytes
def serialize_csv(series: str, data: List[Pair]) -> str:
    """
    >>> data = [Pair(2,3), Pair(5,7)]
    >>> serialize_csv("test", data)
    b'x,y\\r\\n2,3\\r\\n5,7\\r\\n'
    """
    buffer = io.StringIO()
    wtr = csv.DictWriter(buffer, Pair._fields)
    wtr.writeheader()
    wtr.writerows(r._asdict() for r in data)
    return buffer.getvalue()
```

CSV 模块的读取器和写入器混合了命令式元素和函数式元素。我们必须创建写入器，并按照严格的顺序正确创建表头。上面使用了 `Pair` 命名元组的`_fields` 属性来为写入器确定列标题。

写入器的 `writerows()`方法可以接收一个惰性生成器函数。本例中使用了每个 `Pair` 对象的`_asdict()`方法来返回一个适合 CSV 写入器使用的字典。

15.3.6 序列化数据为 XML 格式

下面介绍一个使用 Python 内置的库进行 XML 序列化的方法，这将从单独的标签构建出一个文档。一种常用的替代方法是使用 Python 的自省特性来进行检查并将 Python 对象和类名映射为 XML 标记和属性。

XML 序列化过程如下：

```
import xml.etree.ElementTree as XML

def serialize_xml(series: str, data: List[Pair]) -> bytes:
    """
    >>> data = [Pair(2,3), Pair(5,7)]
    >>> serialize_xml( "test", data )
    b'<series
```

```
    name="test"><row><x>2</x><y>3</y></row><row><x>5</x><y>7</y></row></series>'
    """
    doc = XML.Element("series", name=series)
    for row in data:
        row_xml = XML.SubElement(doc, "row")
        x = XML.SubElement(row_xml, "x")
        x.text = str(row.x)
        y = XML.SubElement(row_xml, "y")
        y.text = str(row.y)
    return cast(bytes, XML.tostring(doc, encoding='utf-8'))
```

这样就创建了一个顶层元素`<series>`，并将子元素`<row>`放在它的下面。在每一个子元素`<row>`内创建了`<x>`标签和`<y>`标签，并将文本内容赋给了各个标签。

使用 `ElementTree` 库作为构建 XML 文档的接口往往是命令式的，因此它不适合函数式设计。除了命令式做法外，请注意并没有创建 DTD（documnet type definition，文档类型定义）或 XSD（XML schemas definition，XML 模式定义），也尚未为标签分配合适的命名空间，同时省略了`<?xml version="1.0"?>`处理指令，而通常它会作为 XML 文档的第一项。

函数 `XML.tostring()`有一个类型提示用于声明返回的是 `str`。这样做通常是正确的，但当提供的是 `encoding` 参数时，结果的类型会变为 `bytes`。并没有简单的方法来根据参数的值判断返回值的类型，因此使用显式的 `cast()`将实际类型告知 mypy。

更高级的序列化库可能会有所帮助，访问 https://wiki.python.org/moin/PythonXml 查看更多选择。

15.3.7　序列化数据为 HTML

序列化的最后一个示例将展示创建 HMTL 文档的复杂性。这种复杂性来自我们期望提供包含大量上下文信息的完整 Web 页面。解决该 HTML 问题的一个方法如下：

```
import string
data_page = string.Template("""\
<html>
<head><title>Series ${title}</title></head>
<body>
<h1>Series ${title}</h1>
<table>
<thead><tr><td>x</td><td>y</td></tr></thead>
<tbody>
${rows}
</tbody>
</table>
</body>
</html>
""")

@to_bytes
```

```
def serialize_html(series: str, data: List[Pair]) -> str:
    """
    >>> data = [Pair(2,3), Pair(5,7)]
    >>> serialize_html("test", data) #doctest: +ELLIPSIS
b'<html>...<tr><td>2</td><td>3</td></tr>\\n<tr><td>5</td><td>7</td></tr>...
    """
    text = data_page.substitute(
        title=series,
        rows="\n".join(
            "<tr><td>{0.x}</td><td>{0.y}</td></tr>".format(row)
        for row in data)
    )
    return text
```

该序列化函数包含两部分。第一部分是一个包含基本 HTML 页面的 `string.Template()` 函数。它有两个占位符，使得数据能插入文档中。方法`${title}`指示了可以插入标题信息的位置，而方法`${rows}`则指示了可以插入数据行的位置。

该函数使用简单的格式化字符串创建各个数据行。它们拼接成一个更长的字符串，然后被替换至模板中。

虽然对于上述示例这种简单情况是可行的，但是对于更复杂的生成数据集来说就不太理想了。有许多更复杂的模板工具可用于创建 HTML 页面，它们大都具备在模板中嵌入循环，以及分离初始序列化函数等功能。更多选择见 https://wiki.python.org/moin/Templating.

15.4 跟踪使用情况

许多公用的 API 都需要使用 API Key（API 密钥）。API 提供者会要求使用者注册并提供电子邮件地址或其他联系信息，之后他们会提供一个 API Key 用于激活该 API。

API Key 可用于验证访问，也可以用于授权特定的功能，还可以用于跟踪使用情况。如果在给定的时间内 API Key 使用过频，还可以用它来限制请求。

在业务模型中有多种变化。例如 API Key 的使用可以是可计费事件。对于其他业务来说，流量必须在达到一定的阈值之后才会被要求付费。

其中重要的一点是 API 使用的不可抵赖性，反之这意味着创建的 API Key 可以用作用户身份验证的凭据。密钥必须难以伪造而易于验证。

创建 API Key 的一种简单方法是使用加密随机数来生成难以预测的密钥字符串。可以用模块 `secrets` 生成唯一的 API Key，并且这些密钥值会分配给客户端用于跟踪使用情况。

```
>>> import secrets
>>> secrets.token_urlsafe(18*size)
'kzac-xQ-BB9Wx0aQoXRCYQxr'
```

针对随机字节码，我们使用 Base64 编码来创建字符序列。以 3 的倍数为长度可以避免 Base64 编码的尾部出现=符号。我们使用了 URL 安全的 Base64 编码，因此生成的字符串中不会包含/或+字符。这意味着密钥可以用作 URL 的一部分，或者包含在报头中。



更细致的加密方法并不会产生更随机的数据，使用 `secrets` 足以确保无法伪造已经分配给其他用户的密钥。

另一种选择是使用 `uuid.uuid4()` 来创建随机的 UUID（universally unique identifier，通用唯一标识符）。它是一个包含 36 个字符的字符串，其中 32 个为十六进制数字，4 个为-标点符号。随机的 UUID 也很难伪造。

还有一种选择是使用 `itsdangerous` 包来创建 JSON 格式的 Web 签名。它使用一个简单的加密系统来确保密钥在对客户端不透明的情况下，依然能为服务器所用。请访问 http://pythonhosted.org/itsdangerous/以获取更多信息。

基于 REST 的 Web 服务器需要一个带有效密钥的小型数据库，还可能需要客户端的一些联系信息。如果一个 API 请求包含了数据库中已有的密钥，则代表请求是由相关用户发起的。如果一个 API 请求未包含一个已知的密钥，则可以通过一个简单的 `401 UNAUTHORIZED` 响应来拒绝该请求。由于密钥本身是一个 24 字符的字符串，因此数据库会非常小，很容易缓存到内存中。

该小型数据库可以是一个简单文件，并由服务器加载后将 API Key 映射至需要授权的权限上。可以在系统启动时和检测修改时间时读取该文件，查看服务器中缓存的版本和当前的版本是否一致。当有新的密钥可用时，该文件会更新，服务器将重新读取该文件。

普通的日志抓取可能足以显示某个特定密钥的使用情况了。更复杂的应用程序可能会在单独的日志文件或数据库中记录 API 请求以简化分析。

15.5　小结

本章讨论了如何将函数式设计应用于通过基于 REST 的 Web 服务来提供内容的问题需求；介绍了 WSGI 标准是如何应用于某些函数式应用程序的；还展示了如何通过从应用程序的函数所使用的请求中提取元素，来将函数式设计嵌入 WSGI 的环境中。

对于简单的服务，问题需求通常可以分解为 3 项操作：获取数据、搜索或过滤，以及对结果进行序列化。可使用 3 个函数来解决这些问题：`raw_data()`、`anscombe_filter()`和 `serialize()`。我们将这些函数封装在一个简单的、兼容 WSGI 的应用程序中，以便将 Web 服务与提取和过滤数据的**实际**处理操作分离开来。

另外介绍了 Web 服务的函数是如何着眼于“幸福之路”并假定所有输入都是有效的。如果输入无效，则普通的 Python 异常处理将抛出异常。WSGI 封装函数会捕获错误并返回相应的状态

码和错误内容。

本章没有讨论与上传数据或接收表单数据来更新持久化数据存储等相关的复杂问题。它们并不比获取数据和序列化结果复杂，但可以通过更巧妙的方式来解决。

对于简答的查询和数据共享，可以使用小型 Web 服务应用程序。可以运用函数式设计模式来确保网站代码简洁明了。对于更复杂的 Web 应用程序，应当考虑使用能正确处理细节的框架。

下一章将介绍一些实用的优化技术，届时会详细介绍第 10 章讲过的`@lru_cache` 装饰器，以及第 6 章提及的其他一些优化技术。

第 16 章

优化与改进

16

本章将介绍一些可以用于创建高性能函数式应用程序的优化技术，主要讨论以下几个主题。

- 扩展第 10 章的`@lru_cache`装饰器的用法。有许多方法可实现记忆化算法。
- 还将介绍如何编写自己的装饰器。更重要的是，将介绍如何使用 `Callable` 对象来缓存记忆化的结果。
- 还会详细介绍第 6 章讲过的一些优化技术，回顾尾递归优化的通用方法。针对某些算法，可以结合记忆化与递归实现来获取良好的性能；而对于其他一些算法，记忆化并非很有帮助，必须寻找其他改进空间。
- 最后将使用 `Fraction` 类来详细展示一个优化精度的案例。

在大多数情况下，对程序的细小改动只能略微提升性能。用匿名函数代替函数对性能影响极小。如果程序慢到难以接受，通常必须找寻全新的算法或数据结构。用 $O(n \log n)$的算法替代 $O(n^2)$的算法是提升性能的最佳方式。

重新设计算法可参考 http://www.algorist.com，其中的资源有助于为特定问题找到更好的算法。

16.1 记忆化和缓存

正如第 10 章所提到的，许多算法可以受益于记忆化。首先回顾一些之前的示例，从而给出那些能获益于记忆化的函数类型特征。

第 6 章介绍了几种常用的递归。最简单的递归形式是尾递归，它的参数易于和缓存中的值匹配。如果参数是整型值、字符串或者实例化的集合，那么可以快速比较参数来确定是否缓存了先前的计算结果。

从这些示例中可以看出，阶乘计算或查找斐波那契数等整型值计算的性能都会显著提升。适用于整型值的数值算法还包括查找质因子和计算整数的幂次方。

斐波那契数的递归版本的计算结果 $F(n)$包含两个尾递归调用，定义如下：

$$F_n = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ F_{n-1}+F_{n-2} & n \geqslant 2 \end{cases}$$

这可以转化成一个循环,但设计上的任何改变都需要考量。递归定义的记忆化版本会非常快,并且无须过多地思考设计。

Syracuse 函数是一类用于计算分形值的函数。Syracuse 函数 $S(n)$定义如下：

$$S(n) = \begin{cases} \dfrac{n}{2} & n \text{ 为偶数} \\ 3n+1 & n \text{ 为奇数} \end{cases}$$

递归调用后，会得到一个从起始值 n 开始的数值链 C：

$$C(n) = [n, S(n), S(S(n)), S(S(S(n))), \cdots] = [S^0(n), S^1(n), S^2(n), S^3(n), \cdots]$$

奇偶归一猜想（Collatz conjecture）是一个**结果总为 1 的 Syracuse 函数**。由 $S(1)$开始的值会形成 1, 4, 2, 1, ···的循环。探索该函数的行为需要记忆化的中间结果。一个有趣的问题是定位极长序列。请参阅 https://projecteuler.net/problem=14，了解有关谨慎使用缓存的问题。

Syracuse 函数的递归应用是一个具有“吸引子”的函数示例，其中数值被“吸引”至 1。在一些高维函数中，吸引子可以是一条线，或是分形曲线。当吸引子是一个点时，记忆化会有所帮助，否则由于每个分形值都是唯一的，记忆化可能会成为障碍。

在处理集合时，缓存的好处可能会消失。如果集合的整型值、字符串或元组的数量碰巧相同，那么集合可能会重复，从而节省时间。然而，如果需要对集合进行多次计算，那么最好手动记忆化：执行一次计算并将结果赋给变量。

在使用可迭代对象、生成器函数和其他惰性对象时，缓存或记忆化是无用的。为了提供源序列中的下一个值，惰性函数会执行最少量的工作。

包含度量的原始数据通常使用浮点数。由于浮点数之间的精确比较可能无法正常工作，因此中间结果的记忆化也可能不起作用。

然而，包含计数的原始数据可能会受益于记忆化。由于它们都是整型值，因此可以确定整型比较（可能）会避免重新计算先前的值。一些应用于计数的统计函数会获益于使用 `fractions` 模块而非浮点数。当用 `Fraction(x,y)`方法代替 `x/y`，便能进行精确值匹配。可以使用 `float(some_fraction)`方法来生成最终结果。

16.2　指定记忆化

记忆化的基本思想十分简单，用@lru_cache 装饰器即可捕获。可以将该装饰器应用于任何函数来实现记忆化。在某些情况下，可以用更特定的方式来改进此通用思想。下面在大量潜在可优化的多值函数中选取一个，并在更复杂的案例研究中考察另一个。

二项式$\binom{n}{m}$表示 n 个不同事物按组大小 m 排列方式的数量，其值如下所示：

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

显然，应该缓存单独计算的阶乘值，而不是重新计算所有的乘积，然而缓存整个二项式计算也可能有用。

下面创建一个包含多个内部缓存的 Callable 对象。需要用到的辅助函数如下：

```
from functools import reduce
from operator import mul
from typing import Callable, Iterable

prod: Callable[[Iterable[int]], int] = lambda x: reduce(mul, x)
```

函数 prod()计算了可迭代数值的乘积。将它定义为了使用*运算符的归约。

以下带有两个缓存的 Callable 对象使用了该 prod()函数。

```
class Binomial:
    def __init__(self):
        self.fact_cache = {}
        self.bin_cache = {}
    def fact(self, n: int) ->int:
        if n not in self.fact_cache:
            self.fact_cache[n] = prod(range(1, n+1))
        return self.fact_cache[n]
    def __call__(self, n: int, m: int) -> int:
        if (n, m) not in self.bin_cache:
            self.bin_cache[n, m] = (
                self.fact(n)//(self.fact(m)*self.fact(n-m)))
        return self.bin_cache[n, m]
```

上面创建了两个缓存：一个用于阶乘值，另一个用于二项式系数值。内部的 fact()方法使用了 fact_cache 属性。如果值不在缓存中，则进行计算并将其添加到缓存中。外部的__call__()方法以类似的方式使用了 bin_cache 属性：如果已经计算过了特定二项式，则直接返回该值；如果没有，则会用内部的 fact()方法计算一个新值。

可以如下所示使用上述 Callable 类：

```
>>> binom = Binomial()
>>> binom(52, 5)
2598960
```

这显示了如何从类中创建一个 `Callable` 对象，然后在特定的一组参数上调用该对象。将 52 张牌派分为一手 5 张牌，则会有 260 万种可能。

16.3 尾递归优化

第 6 章等处介绍了如何将简单的递归优化为 `for` 循环。以如下阶乘的简单递归定义为例：

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n \neq 0 \end{cases}$$

优化递归的通用方法如下。

- 设计递归。这意味着可以测试基本情形和简单函数形式的递归情形，尽管慢但结果是正确的。简单定义如下：

```
def fact(n: int) -> int:
    if n == 0: return 1
    else: return n*fact(n-1)
```

- 如果递归的最后有一个简单的调用，则用 `for` 循环替代递归，其定义如下：

```
def facti(n: int) -> int:
    if n == 0: return 1
    f = 1
    for i in range(2,n):
        f = f*i
    return f
```

当递归出现在一个简单函数的最后，则将它描述为尾调用优化。许多编译器会将其优化成一个循环。Python 没有优化编译器，不会进行这种尾调用转换。

这种模式很常用。执行尾调用优化可以提升性能，并突破可执行递归数量的上限。

在进行任何优化前，函数必须能正常运作。对此，一个简单的 `doctest` 字符串通常就足够了。可以对阶乘函数做如下注释：

```
def fact(n: int) -> int:
    """Recursive Factorial
    >>> fact(0)
    1
    >>> fact(1)
    1
    >>> fact(7)
    5040
    """
```

```
    if n == 0: return 1
    else: return n*fact(n-1)
```

这样就添加了两个边缘情况：显式的基本情形和基本情形后的第一项。还添加了一个涉及多次迭代的项，让我们得以调整代码。

当有更复杂的函数组合时，可能需要执行以下命令：

```
binom_example = """
>>> binom = Binomial()
>>> binom(52, 5)
2598960
"""

__test__ = {
    "binom_example": binom_example,
}
```

函数 `doctest.testmod()`使用了变量`__test__`。字典中所有和变量`__test__`相关联的值都用于了 `doctest` 字符串。这是测试复合函数特性的一种简便方法。由于它测试了多个软件组件的集成性，因此也称其为**集成测试**。

一组可用的测试代码会利于优化，并便于确认优化的正确性。下面是一句描述优化的名言：

> “让程序错上加错并不是罪。”
>
> ——Jon Bentley

这句话出自《编程珠玑（续）》一书的“计算机科学箴言集”章。重要的是应该只优化正确的代码。

16.4　优化存储

优化没有通用规则。我们通常关注性能优化，因为可以使用一些工具（例如**大 O** 复杂性度量）来判断算法能否有效地解决给定的问题。通常单独处理存储优化：可以研究算法的流程，并估计不同存储结构所需的存储空间。

在许多情况下，这两种考量是相互对立的。在某些情况下，一个性能非常好的算法可能需要大型数据结构。如果不能大幅增加所需的存储空间，就无法扩展这种算法。我们的目标是设计一种速度快且所用的存储量可接受的算法。

我们可能需要花时间研究算法的替代方法，从而找到一种能合理平衡时空（space-time tradeoff）的方法。在维基百科的链接里可以找到一些通用的优化技术。

Python 中的一种内存优化技术是使用可迭代对象。它具有适当实例化集合的一些属性，但不

一定占用存储空间。不适用于可迭代对象的运算很少（如 `len()` 函数）。对于其他运算，节省内存的特性使得程序可以处理非常大的集合。

16.5 优化精度

少数情况下，需要优化计算的精度。这有一定的挑战性，而且可能需要一些相当高级的数学运算来确定给定方法的精度限制。

在 Python 中，可以用 `fractions.Fraction` 值代替浮点近似。对于某些应用，这样做可以得到比浮点数更精确的解，因为相比浮点尾数，分子和分母使用了更多比特。

使用 `decimal.Decimal` 值来处理货币十分重要。一种常见的错误是使用 `float` 值。当使用 `float` 值时，由于作为输入的 `Decimal` 值与浮点数使用的二进制近似之间的不匹配，从而引入了额外的噪声位。使用 `Decimal` 值可以防止引入微小的误差。

在许多情况下，可以对 Python 程序进行小的改动，将 `float` 值转换为 `Fraction` 或 `Decimal` 值。在处理超越函数时，这种改动不一定有用。超越函数的定义涉及无理数。

根据用户需求降低精度

对于某些计算，分数值可能比浮点数更直观，这是以一种用户能理解并做出处理的方式来展示统计结果的一部分。

例如，卡方检验通常涉及计算实际值和预期值之间的 χ^2 比较。随后可以用这个比较值来测试 χ^2 的累积分布函数。当预期值和实际值没有特定关系（也称“空关系”）时，变化将是随机的，这个值往往很小。当接受了零假设后，需要在其他地方寻找它们的关系。当实际值与期望值区别较大时，我们可能会拒绝零假设，而进一步探索以确定两者关系的准确性质。

对于选定的 χ^2 值和给定的自由度，决策通常基于 χ^2 **累积分布函数**（cumulative distribution function，CDF）。尽管表格中的 CDF 值大多数是无理数，但通常不会保留超过 2 个或 3 个小数。它只是一个决策工具，0.049 和 0.05 在实际意义上没有区别。

拒绝零假设广泛使用的概率是 0.05，这是一个小于 1/20 的 `Fraction` 对象。在向用户展示数据时，可以将结果描述为分数形式，而像 0.05 这样的值不怎么直观。用 1/20 来描述关联的几率有助于人们感知这种关联的可能性。

16.6 案例研究：卡方决策

下面介绍一个常见的统计决策。在 http://www.itl.nist.gov/div898/handbook/prc/section4/prc45.htm 中有对该决策的详细描述。

这是一个关于数据是否随机分布的卡方决策。为了做出这个决策，需要计算一个预期分布，并将观察到的数据与预期进行比较。相差较大意味着需要进一步研究。相差不大意味着可以使用零假设，因为没什么值得研究了，即这些差异仅仅是随机变化造成的。

下面介绍如何使用 Python 来处理数据。首先介绍一些不属于案例研究的背景知识，但常出现在 EDA 应用程序中。需要收集原始数据并生成有用的可供分析的汇总信息。

在生产质量保障过程中，将有硅片缺陷的数据收集到数据库中。可以使用 SQL 查询来提取缺陷细节供后续分析。例如查询语句可能如下所示：

```
SELECT SHIFT, DEFECT_CODE, SERIAL_NUMBER FROM some tables;
```

该查询的输出将是带有各个缺陷详情的.csv 文件。

```
shift,defect_code,serial_number
1,None,12345
1,None,12346
1,A,12347
1,B,12348
and so on. for thousands of wafers
```

需要汇总先前的数据，可以在 SQL 查询层面使用 COUNT 语句和 GROUP BY 语句进行汇总，也可以在 Python 应用层面进行汇总。尽管通常认为纯数据库汇总更高效，但并非总是如此。在某些情况下，对原始数据的简单提取和用 Python 程序进行汇总可能比 SQL 汇总更快。如果看重性能，那么必须衡量这两种方法，而不是设想数据库操作总是最快的。

在某些情况下，可以高效地从数据库中获取汇总数据。汇总必须包含三个属性：轮换（shift）、缺陷类型和观测到的缺陷数量。汇总数据如下所示：

```
shift,defect_code,count
1,A,15
2,A,26
3,A,33
and so on.
```

输出会显示轮换和缺陷类型的所有 12 种组合。

稍后将详细介绍如何读取原始数据并创建汇总。这便是 Python 的强大之处：处理原始源数据。

需要观察并比较轮换和缺陷个数的总体预期。如果观测到的数量和预期数量间的差异可以归因于随机波动，便要接受零假设，即没有什么错误点值得关注。如果这些数字不符合随机变化，那么就有问题需要进一步研究。

16.6.1 使用 Counter 对象过滤和约分原始数据

我们把基本的缺陷计数表示为 collections.Counter 参数。下面将根据原始数据中的轮

换和缺陷类型来构造缺陷计数。以下代码从.csv 文件中读取了一些原始数据。

```
from typing import TextIO
import csv
from collections import Counter
from types import SimpleNamespace

def defect_reduce(input_file: TextIO) -> Counter:
    rdr = csv.DictReader(input_file)
    assert set(rdr.fieldnames) == set(
        ["defect_type", "serial_number", "shift"])
    rows_ns = (SimpleNamespace(**row) for row in rdr)
    defects = (
        (row.shift, row.defect_type)
        for row in rows_ns if row.defect_type)
    tally = Counter(defects)
    return tally
```

上述函数会基于通过 input 参数提供的打开文件来创建一个字典读取器。我们确认了列名需和 3 个预期的列名相匹配。在某些情况下，文件会包含额外必须被忽略的列，此时断言会类似于 set(rdr.fieldnames) <= set([...])，用于确认实际的列名是所需列的子集。

我们为每一行创建了一个 types.SimpleNamespace 参数。文件中的列名都是有效的 Python 变量名，便于将字典转换为命名空间对象，使得我们可以用稍简单的语法来引用行中的项。具体而言，后面的生成器表达式使用 row.shift 和 row.defect_type 而不是 row['shift'] 和 row['defect_type']作为引用。

可以使用更复杂的生成器表达式来组合映射和过滤。过滤每一行，以忽略那些没有代码缺陷的行。对于有代码缺陷的行，可以映射一个表达式，基于 row.shift 和 row.defect_type 引用创建一个二元组。

在某些应用程序中，过滤器不会是 row.defect_type 这样的简单表达式，可能需要编写更复杂的条件语句。在这种情况下，使用 filter()函数将复杂条件应用于提供数据的生成器表达式可能会有帮助。

给定一个生成(shift, defect)元组序列的生成器，可以通过从生成器表达式中创建一个 Counter 对象来汇总它们。创建该 Counter 对象来处理惰性生成器表达式，该表达式会读取源文件、从行中提取字段、过滤行并汇总计数。

使用 defect_reduce()函数来收集和汇总数据，如下所示：

```
with open("qa_data.csv") as input:
    defects = defect_reduce(input)
print(defects)
```

我们可以打开一个文件，收集并显示缺陷信息，以确保正确地汇总了轮换和缺陷类型。由于结果是一个 Counter 对象，因此如果有其他源数据，就可以将它与其他 Counter 对象结合起来了。

值 defects 如下所示：

```
Counter({('3', 'C'): 49, ('1', 'C'): 45, ('2', 'C'): 34,
 ('3', 'A'): 33, ('2', 'B'): 31, ('2', 'A'): 26,
 ('1', 'B'): 21, ('3', 'D'): 20, ('3', 'B'): 17,
 ('1', 'A'): 15, ('1', 'D'): 13, ('2', 'D'): 5})
```

通过轮换('1', '2', or '3')和缺陷类型（从'A'到'D'）组织了缺陷计数。下面介绍汇总数据的其他一些输入，能体现数据在汇总级别的常规用法。

一旦读取了数据，下一步就是生成两个概率，这样就可以正确地计算每个轮换和每种缺陷的预期缺陷了。我们不想把总缺陷数除以 12，因为这并不能反映实际的轮换或者缺陷类型。轮换可能或多或少具有相同效果，但缺陷频率肯定不会相似。我们期望一些缺陷是非常罕见的，另一些则更为常见。

16.6.2　读取汇总信息

作为读取所有原始数据的替代方法，可以只考虑处理汇总计数。我们希望创建一个类似于之前示例的 Counter 对象，它会以轮换班和缺陷代码作为键，以缺陷计数作为值。对于给定的汇总信息，只需从输入字典中创建一个 Counter 对象。

负责读取汇总数据的函数如下：

```
from typing import TextIO
from collections import Counter
import csv
def defect_counts(source: TextIO) -> Counter:
    rdr = csv.DictReader(source)
    assert set(rdr.fieldnames) == set(
        ["defect_type", "serial_number", "shift"])
    rows_ns = (SimpleNamespace(**row) for row in rdr)
    convert = map(
        lambda d: ((d.shift, d.defect_code), int(d.count)),
        rows_ns)
    return Counter(dict(convert))
```

需要一个打开的文件作为输入。首先创建一个 csv.DictReader()函数，来解析从数据库中获得的原始 CSV 数据。其中包含了一条 assert 语句来保证文件包含了预期的数据。

该变体使用匿名对象为每行创建一个二元组。这些二元组具有从轮换和缺陷代码构建而来的复合键，以及整型转换后的计数。生成的结果是一个类似于((shift,defect), count), ((shift,defect), count), ...)的序列。当把 lambda 映射到 row_ns 生成器后，会得到一个能生成二元组序列的生成器函数。

然后从二元组集合创建一个字典，并使用这个字典构建一个 Counter 对象。这个 Counter 对象易于和其他 Counter 对象组合在一起，这让我们可以结合从多个数据源获取的汇总详情。

该示例中只有一个数据源。

可以将这个单一源赋给 defects 变量，其值如下所示：

```
Counter({('3', 'C'): 49, ('1', 'C'): 45, ('2', 'C'): 34,
 ('3', 'A'): 33, ('2', 'B'): 31, ('2', 'A'): 26,
 ('1', 'B'): 21, ('3', 'D'): 20, ('3', 'B'): 17,
 ('1', 'A'): 15, ('1', 'D'): 13, ('2', 'D'): 5})
```

这与此前显示的详情汇总相符。不同的是，此时已经汇总了原始数据。这种情况通常出现在从数据库提取了数据并使用 SQL 进行分组操作的时候。

16.6.3 Counter 对象的求和计算

我们需要根据轮换和类型计算缺陷的概率。为了计算预期的概率，需要首先进行简单的求和。下面对所有缺陷的值进行求和，可以通过执行以下命令计算得到：

```
total = sum(defects.values())
```

这是直接从赋给 defects 变量的 Counter 对象中计算得到的，会显示样本集中共有 309 个缺陷。

还需要通过轮换和类型来获取缺陷，这意味着需要从原始缺陷数据中提取两种子集。**按轮换**提取将只使用 Counter 对象中(shift,defect type)键的前半部分，**按类型**提取则使用键值对的后半部分。

可以从赋值给 defects 变量的初始 Counter 对象集合中提取并创建额外的 Counter 对象来进行汇总。按轮换汇总如下所示：

```
shift_totals = sum(
    (Counter({s: defects[s, d]}) for s, d in defects),
    Counter()  # start value = empty Counter
)
```

这样就创建了单独的 Counter 对象集合。这些对象都具有一个轮换 s 作为键，以及相应的缺陷计数 defects[s,d]。生成器表达式会创建 12 个这样的 Counter 对象用于从 4 种缺陷和 3 种轮换的所有组合中提取数据。可以使用 sum()函数来组合 Counter 对象，以此获得按轮换组织的 3 组汇总信息。



对于 sum()函数，不能使用默认的初始值 0，必须提供一个空的 Counter()对象作为初始值。

创建类型汇总的表达式类似于创建轮换汇总的表达式。

```
type_totals = sum(
    (Counter({d: defects[s, d]}) for s, d in defects),
```

```
    Counter()  # start value = empty Counter
)
```

使用缺陷类型 d 而不是轮换类型作为键，创建了十余个 Counter 对象，否则结果会一样。

轮换总计如下：

```
Counter({'3': 119, '2': 96, '1': 94})
```

缺陷类型总计如下：

```
Counter({'C': 128, 'A': 74, 'B': 69, 'D': 38})
```

将汇总结果保存为 Counter 对象，而不是创建简单的 dict 对象，甚至 list 实例。之后通常会将它们作为普通字典来使用，然而在某些情况下，需要合适的 Counter 对象而非精简的字典对象。

16.6.4　Counter 对象的概率计算

前面在两个独立的过程中读取了数据并计算了汇总信息。有时需要在读取初始化数据时创建汇总信息，这种优化能节省处理时间。可以编写一个更复杂的输入约分操作，来计算所有总数、轮换总数和缺陷类型总数。每次将这些 Counter 对象构建为一项。

前面重点介绍了使用 Counter 实例，因为它们很灵活。对于数据采集的任何更改仍然会创建出 Counter 实例，并且不会更改后续的分析。

按轮换和按缺陷类型计算的缺陷概率如下：

```
from fractions import Fraction
P_shift = {
    shift: Fraction(shift_totals[shift], total)
    for shift in sorted(shift_totals)
}
P_type = {
    type: Fraction(type_totals[type], total)
    for type in sorted(type_totals)
}
```

这样就创建了两个映射：P_shift 和 P_type。字典 P_shift 将轮换映射到一个表示轮换在总缺陷个数中占比的 Fraction 对象。类似地，字典 P_type 将缺陷类型映射到一个表示类型在总缺陷个数中占比的 Fraction 对象。

选用了 Fraction 对象来保留所有输入值的精度，因为在处理这样的计数时，我们希望得到更直观的概率值，方便人们查看数据。

数据 P_shift 的值如下所示：

```
{'1': Fraction(94, 309), '2': Fraction(32, 103),
 '3': Fraction(119, 309)}
```

数据 P_type 的值如下所示：

```
{'A': Fraction(74, 309), 'B': Fraction(23, 103),
 'C': Fraction(128, 309), 'D': Fraction(38, 309)}
```

对于一些人来说，32/103 或 96/309 这样的值比 0.3106 更有意义。从 Fraction 对象中获得 float 值很容易，稍后将展示这一点。

在 Python 3.6 中，字典中的键将保留源数据中键的顺序。在之前的 Python 版本中，键的顺序是不可预测的。在这个示例中，顺序无关紧要，但当键的顺序可预测时，会有助于调试工作。

所有轮换似乎与缺陷产出处于同一水平。缺陷类型变化无常，这是典型的情况。缺陷 C 似乎是相对常见的问题，缺陷 B 则不那么常见，也许第二个缺陷在更复杂的情况下才会出现。

16.7　计算期望值并显示列联表

预期的缺陷产出是一个组合概率。下面将计算轮换缺陷与缺陷类型概率的乘积，为此需要计算轮换和缺陷类型组合的所有 12 种概率。可以对观测到的数字进行加权，并计算缺陷的详细预期。

计算期望值的代码如下所示：

```
expected = {
    (s, t): P_shift[s]*P_type[t]*total
    for t in P_type
    for s in P_shift
}
```

我们会创建一个与 defectsCounter 对象相似的字典。该字典会有一个带有键值的二元组序列，其中键是轮换和缺陷类型的二元组。字典是通过一个生成器表达式构建而来的，它显式枚举了 P_shift 和 P_type 字典中所有键的组合。

字典 expected 的值如下所示：

```
{('2', 'B'): Fraction(2208, 103),
 ('2', 'D'): Fraction(1216, 103),
 ('3', 'D'): Fraction(4522, 309),
 ('2', 'A'): Fraction(2368, 103),
 ('1', 'A'): Fraction(6956, 309),
 ('1', 'B'): Fraction(2162, 103),
 ('3', 'B'): Fraction(2737, 103),
 ('1', 'C'): Fraction(12032, 309),
 ('3', 'C'): Fraction(15232, 309),
 ('2', 'C'): Fraction(4096, 103),
```

```
 ('3', 'A'): Fraction(8806, 309),
 ('1', 'D'): Fraction(3572, 309)}
```

映射的每一项都以轮换和缺陷类型作为键，且它与一个 `Fraction` 值相关联，这个值基于轮换次数的缺陷概率，以及缺陷类型乘以总缺陷次数的缺陷概率。一些分数约分了，例如值 6624/309 可以简化为 2208/103。

大的数不适合用分数表示，将其呈现为 `float` 值通常更容易。小数值（如概率）有时用分数表示则更易于理解。

然后将成对输出观测到的次数和预期的次数，这有助于可视化数据。我们将创建如下内容来汇总观测到的值和预期的值：

```
obs exp    obs exp     obs exp    obs exp
 15 22.51    21 20.99    45 38.94    13 11.56    94
 26 22.99    31 21.44    34 39.77     5 11.81    96
 33 28.50    17 26.57    49 49.29    20 14.63   119
 74          69         128          38         309
```

这里显示了 12 个单元格。每个单元格的值都包含观测到的缺陷数量和预期的缺陷数量。每一行的最后是轮换总数，每一列的最下面是缺陷总数。

在某些情况下，可以将这种数据导出为 CSV 格式并构建一个电子表格。在其他一些情况下，可以构建一个 HTML 版本的列联表，并将布局细节留给浏览器去处理。这里显示的是纯文本版本。

以下代码包含的一系列语句用于创建如前所示的列联表：

```
print("obs exp "*len(type_totals))
for s in sorted(shift_totals):
    pairs = [
        f"{defects[s,t]:3d} {float(expected[s,t]):5.2f}"
        for t in sorted(type_totals)
    ]
    print(f"{' '.join(pairs)} {shift_totals[s]:3d}")
footers = [
    f"{type_totals[t]:3d} "
    for t in sorted(type_totals)]
print(f"{' '.join(footers)} {total:3d}")
```

这样会将缺陷类型展开成一行。前面已经编写了足够多的 `obsexp` 列标题来涵盖所有缺陷类型。对于每个轮换，会生成一行观测值和实际值的配对，并在后面加上总的轮换数。底部将生成一行包含缺陷类型总数和总计数量的脚注。

这样的列联表有助于可视化对观测值和期望值的比较。可以计算这两组值的卡方值，以便于我们确定数据是随机的或是值得进一步研究。

16.7.1 计算卡方值

值 χ^2 基于 $\sum_i \frac{(e_i - o_i)^2}{e_i}$，其中 e 是预期值，o 是观测到的值。示例中有两个维度：轮换 s 和缺陷类型 t，因此最终可以表示为 $\chi^2 = \sum_s \sum_t \frac{(e_{st} - o_{st})^2}{e_{st}}$。

可以如下所示计算该特定公式的值：

```
diff = lambda e, o: (e-o)**2/e

chi2 = sum(
    diff(expected[s, t], defects[s, t])
    for s in shift_totals
    for t in type_totals
)
```

这样就定义了一个小型匿名函数来优化计算。这使得只需执行一次 `expected[s,t]`和`defects[s,t]`属性的计算，即便两处用到了期望值。对于这个数据集，最终的 χ^2 值为 19.18。

由于有 3 个轮换和 4 种缺陷类型，因此自由度一共为 6。由于我们认为它们是彼此独立的，所以会得到$(3-1) \times (4-1) = 6$。根据卡方表显示，任何低于 12.5916 的数据都有 1/20 的概率是完全随机的。由于结果是 19.18，因此数据不太可能是随机的。

累积分布函数表明值 19.18 的概率值为 0.00387，相当于有 4/1000 的概率是随机的。完整分析的下一步是设计一个后续研究，用于发现不同缺陷类型和轮换的详细特性。需要找到与缺陷相关性最大的自变量，进而继续分析。这项工作的合理性在于 χ^2 的值表明结果并不是简单的随机变化。

一个补充问题是关于阈值 12.5916 的。可以在统计表中找到这个值，也可以直接计算这个阈值。这会引出许多函数式编程的有趣示例。

16.7.2 计算卡方阈值

χ^2 测试的本质是一个阈值，它基于自由度个数和我们愿意接受或拒绝零假设的不确定性。通常建议使用 0.05（1/20）左右的阈值来拒绝零假设。我们希望数据只有 1/20 的概率是随机而有意义的。换言之，我们希望数据有 19/20 的概率属于简单的随机变化。

由于卡方值的计算涉及许多超越函数，因此通常呈现为表格形式。在某些情况下，软件库会提供 χ^2 累积分布函数的实现，这使得我们可以计算该值而不用在表格中查找它。

对于一个 χ^2 值 x 和自由度 f，其累积分布函数定义如下：

$$F(x;k)=\frac{\gamma\left(\frac{k}{2},\frac{x}{2}\right)}{\Gamma\left(\frac{k}{2}\right)}$$

通常用 $p=1-F(\chi^2;k)$表示随机概率，即：如果 $p>0.05$，则数据可以理解为随机的，此时零假设成立，否则数据不太可能是随机的，有必要进一步研究。

累积分布为不完全伽马函数 $\gamma(s,z)$和完全伽马函数 $\Gamma(x)$之比。计算这些函数值的通用方法可能涉及一些相当复杂的数学计算。我们取捷径巧妙地实现了两个非常好的近似，从而可以重点关注这个问题本身。这两个函数有助于研究函数式设计中的一些额外问题。

这两个函数都需要进行阶乘计算 $n!$。前面介绍过过许多阶乘计算形式了，这里使用如下版本：

```
@lru_cache(128)
def fact(k: int) -> int:
    if k < 2:
        return 1
    return reduce(operator.mul, range(2, int(k)+1))
```

这便是$k!=\prod_{2\leqslant i\leqslant k} i$，它代表 2 到 k（含 K）的乘积。该实现不涉及递归。由于 Python 中的整型值可能非常大，因此这个值的计算并没有实际上限。

16.7.3　计算不完全伽马函数

不完全伽马函数具有级数展开式，这意味着需要计算一系列的值然后对它们进行求和。更多相关信息，请访问 http://dlmf.nist.gov/8。

$$\gamma(s,z)=\sum_{0\leqslant k<\infty}\frac{(-1)^k}{k!}\frac{z^{s+k}}{s+k}$$

这个级数有无穷多个项，这些值最终会变得很小以至于相关性不大。可以建立一个极小值ε，并在下一项小于该值时停止计算。

$(-1)^k$的计算会产生交替的符号：

$$(-1)^0,(-1)^1,(-1)^2,(-1)^3,\cdots=1,-1,1,-1,\cdots$$

当 $s=1$ 且 $z=2$ 时，序列中的项如下所示：

$$2/1,-2/1,4/3,-2/3,4/15,-4/45,\cdots,-2/638512875$$

在某一时刻，额外增加的每一项都不会显著影响结果。

当回顾累积分布函数 $F(x;k)$时，可以考虑使用 fractions.Fraction 值。自由度 k 除以 2 后是一个整数。值 x 既可以是 Fraction，也可以是 float 值，很少会是整型值。

在计算 $\gamma(s, z)$中的项时，值 $\frac{(-1)^k}{k!}$ 会包含整数，并且可以用一个合适的 Fraction 值来表示。然而，整个表达式并不总是一个 Fraction 对象。值 z^{s+k}可以是 Fraction 或 float 值。如果 $s+k$ 不是整数，则会导致结果为无理数。对于无理数，可以用形式较为复杂的 Fraction 对象来近似表示它们的值。

在伽马函数中使用 Fraction 值是可行的，但似乎帮助不大。然而对于完全伽马函数，Fraction 对象具有潜在优势。鉴于此，即使 Fraction 对象能得到无理数值的近似，我们还是会在此实现中使用该对象。

对上面阐述的级数展开的实现如下所示：

```
from typing import Iterator, Iterable, Callable, cast
def gamma(s: Fraction, z: Fraction) -> Fraction:

    def terms(s: Fraction, z: Fraction) -> Iterator[Fraction]:
        """Terms for computing partial gamma"""
        for k in range(100):
            t2 = Fraction(z**(s+k))/(s+k)
            term = Fraction((-1)**k, fact(k))*t2
            yield term
        warnings.warn("More than 100 terms")

    def take_until(
            function: Callable[..., bool], source: Iterable
        ) -> Iterator:
        """Take from source until function is false."""
        for v in source:
            if test(v):
                return
        yield v

ε = 1E-8
g = sum(take_until(lambda t: abs(t) < ε, terms(s, z)))
# sum() from Union[Fraction, int] to Fraction
return cast(Fraction, g)
```

这样就定义了一个 term()函数用于生成一系列项，其中限制了 for 语句最多生成 100 项。也可以使用 itertools.count()函数来生成无限项序列，但这里使用带上限的循环似乎更简单一些。

上面还计算了一个可能是无理数的 z^{s+k}，并基于这个值创建了一个 Fraction 值。除法 $\frac{z^{s+k}}{s+k}$ 会涉及两个 Fraction 相除，并生成一个新的 Fraction 值赋给变量 t2。之后变量 term 的值

是这两个 Fraction 对象的乘积。

此外定义了一个 take_until()的函数，它从一个可迭代对象中取值，直到给定的函数为真时结束。一旦该函数为真，可迭代对象便不再生成新值。上面还定义了一个较小的阈值ε：10^{-8}。我们会从 term()函数中取值，直到取到的值小于ε。这些值的和是对伽马函数部分的近似。

请注意，阈值变量是希腊字母ε。Python 3 允许变量名使用任何 Unicode 字符。

可以用如下测试用例验证计算结果是否正常：

- $\gamma(1, 2) = 1 - \mathrm{e}^{-2} \approx 0.8646647$
- $\gamma(1, 3) = 1 - \mathrm{e}^{-3} \approx 0.9502129$
- $\gamma\left(\frac{1}{2}, 2\right) = \sqrt{\pi} \times \mathrm{erf}(\sqrt{2}) \approx 1.6918067$

误差函数 erf()可以在 Python 的 math 库中找到，不需要专门对此进行近似。

我们重点关注卡方分布。出于一些数学上的考量，通常对不完全伽马函数不感兴趣，因此可以将测试用例缩小到要使用的数值类型上，还可以限制结果的精度。大部分卡方测试的精度取到 3 位小数。我们在测试数据中显示了 7 位小数，这可能多于实际需要。

16.7.4　计算完全伽马函数

与不完全伽马函数相比，完全伽马函数更难实现，其中有多种近似方法。更多相关信息，请访问 http://dlmf.nist.gov/5。Python 的 math 库中有一个可用的版本，它实现了一个广泛适用的近似。

我们对完全伽马函数的完整通用实现并不感兴趣，只关心两种特殊情况：整型值和二分值。对于这两种特殊情形，可以得到精确的解，而不需要依赖近似。

对于整数值，$\Gamma_n = (n-1)!$。整数的完全伽马函数可以依赖此前定义的阶乘函数。

对于二分值，有一个特殊形式：$\Gamma\left(\frac{1}{2}+n\right) = \frac{(2n)!}{4^n n!}\sqrt{\pi}$。它包含一个无理数$\sqrt{\pi}$，因此只能使用 float 或 Fraction 对象来近似表示。

如果使用合适的 Fraction 值，那么可以用以下几个简单用例来设计一个函数：一个 integer 数值、一个分母为 1 的 Fraction 值和一个分母为 2 的 Fraction 值。可以如下所示使用 Fraction 值：

```
sqrt_pi = Fraction(677_622_787, 382_307_718)

from typing import Union
```

```
def Gamma_Half(
        k: Union[int, Fraction]
    ) -> Union[int, Fraction]:
    if isinstance(k, int):
        return fact(k-1)
    elif isinstance(k, Fraction):
        if k.denominator == 1:
            return fact(k-1)
        elif k.denominator == 2:
            n = k-Fraction(1, 2)
            return fact(2*n)/(Fraction(4**n)*fact(n))*sqrt_pi
    raise ValueError(f"Can't compute Γ({k})")
```

将这个函数称为 Gamma_Half 是为了强调它只适用于所有整数二分数。对于整型值，可使用前面定义的 fact()函数。对于分母为 1 的 Fraction 对象，可使用同一个 fact()定义。

如果分母为 2，可以使用更复杂的闭形式值。我们对值 $4^n n!$显式使用了一个 Fraction()函数，还为无理数 $\sqrt{\pi}$ 提供了一个 Fraction 近似。

一些测试用例如下所示：

- ❑ $\Gamma(2)=1$
- ❑ $\Gamma(5)=24$
- ❑ $\Gamma\left(\frac{1}{2}\right)=\sqrt{\pi}\approx 1.774539\approx\frac{582\,540}{328\,663}$
- ❑ $\Gamma\left(\frac{3}{2}\right)=\frac{\sqrt{\pi}}{2}\approx 0.8862269\approx\frac{291\,270}{328\,663}$

也可以用合适的 Fraction 值来表示它们。无理数的表示（平方根和 π）往往会生成数值很大且可读性很差的分数。可以使用更易读的分数形式，如下所示：

```
>>> g = Gamma_Half(Fraction(3, 2))
>>> g.limit_denominator(2_000_000)
Fraction(291270, 328663)
```

这里给出的值，限制了其分母小于 200 万，这样就得到了易读的 6 位数值，可以用它们来进行单元测试。

16.7.5 计算随机分布的概率

有了不完全伽马函数 gamma 和完全伽马函数 Gamma_Half，就可以计算 χ^2的 CDF 值了。该值表示一个给定值是随机还是具有某种相关性。

函数本身十分短小：

```
def cdf(x: Union[Fraction, float], k: int) -> Fraction:
    """X2 cumulative distribution function.
    :param x: X2 value, sum (obs[i]-exp[i])**2/exp[i]
        for parallel sequences of observed and expected values.
    :param k: degrees of freedom >= 1; often len(data)-1
    """
    return (
        1 -
        gamma(Fraction(k, 2), Fraction(x/2)) /
        Gamma_Half(Fraction(k, 2))
    )
```

该函数包含了一些 `docstring` 注释用于解释参数。我们通过自由度和卡方值 x 创建了合适的 `Fraction` 对象。参数 x 可以是 `float` 值或者 `Fraction` 对象，这种灵活性可用于匹配那些完全使用浮点近似的示例。

可以使用 `Fraction(x/2).limit_denominator(1000)`将 `x/2Fraction` 方法的大小限制为相对少量的数字。这会计算出一个正确的 `CDF` 值，而不是包含几十位数的庞大分数。

以下是从 χ^2 表中调取的一些示例数据。更多相关信息，可参考维基百科词条 chi-squared distribution。

执行以下命令以计算正确的 `CDF` 值：

```
>>> round(float(cdf(0.004, 1)), 2)
0.95
>>> cdf(0.004, 1).limit_denominator(100)
Fraction(94, 99)
>>> round(float(cdf(10.83, 1)), 3)
0.001
>>> cdf(10.83, 1).limit_denominator(1000)
Fraction(1, 1000)
>>> round(float(cdf(3.94, 10)), 2)
0.95
>>> cdf(3.94, 10).limit_denominator(100)
Fraction(19, 20)
>>> round(float(cdf(29.59, 10)), 3)
0.001
>>> cdf(29.59, 10).limit_denominator(10000)
Fraction(8, 8005)
```

在给定 χ^2 和多个自由度的情况下，`CDF` 函数生成的值与广泛使用的表格中的值一致。第一个示例展示了 1 个自由度时 χ^2 为 0.004 的概率。第二个示例显示了 1 个自由度时 χ^2 为 10.38 的概率。小的 χ^2 值意味着预期结果和观测结果几乎没有差别。

下面是 χ^2 表格中的一整行，由一个简单的生成器表达式计算得到：

```
>>> chi2 = [0.004, 0.02, 0.06, 0.15, 0.46, 1.07, 1.64, ...2.71, 3.84, 6.64, 10.83]
>>> act = [round(float(x), 3)
...     for x in map(cdf, chi2, [1]*len(chi2))]
>>> act
[0.95, 0.888, 0.806, 0.699, 0.498, 0.301, 0.2, 0.1, 0.05, 0.01, 0.001]
```

这些值显示了在 1 个自由度下，给定 χ^2 和结果之间的相对似然度。与已发布的结果相比，计算结果在第三位小数上有一些细微差异。这意味着可以使用 CDF 计算结果代替在标准统计参考中查找到的 χ^2 值。

函数 CDF() 给出了随机得到的 χ^2 的概率值。

从已发布的表中可以看出，6 个自由度下概率 0.05 的 χ^2 值为 12.5916。该 CDF() 函数的输出如下，显示出与已发布结果较为一致：

```
>>> round(float(cdf(12.5916, 6)), 2)
0.05
```

回顾先前示例，当时算出 χ^2 的实际值是 19.18。该值为随机的概率是：

```
>>> round(float(cdf(19.18, 6)), 5)
0.00387
```

当分母限制为 1000 时，概率为 3/775，说明这些数据不太可能是随机的。这意味着我们可以拒绝零假设，并通过更多的分析来确定造成差异的可能原因。

16.8 函数式编程设计模式

函数式编程有许多常用的设计模式，都是能在各种场景中使用的典型函数式编程方法。

请注意它与面向对象设计模式的主要区别。许多面向对象设计模式旨在使状态管理更加明确，或者协助构建复杂的紧急行为。函数式设计模式的重在从简单形式创建出复杂行为。

本书介绍了许多常用的函数式设计方法，其中大多数尚未给出专有的名称或来历。本节将回顾其中一些模式。

- **柯里化**：可以在偏函数应用中调用它，并由 functools 模块中的 partial() 函数来实现，其思想是基于现有的函数并加上一些（并非全部）函数参数来创建新函数。
- **闭包**：在 Python 中，很容易定义一个返回另一个函数的函数。当返回的函数包含外部函数绑定的变量时，它就是一个闭包。通常在函数返回匿名对象或生成器表达式时完成。也可以将它作为创建参数化装饰器的一部分来实现。
- **纯函数**：常用的无状态函数。在 Python 中，还可以使用非纯函数来处理有状态的输入和输出。此外，系统服务和随机数生成器属于非纯函数的例子。好的函数式设计往往强调尽量使用纯函数，而避免使用 global 语句。
- **函数式复合**：itertools 库包含了许多函数式复合的工具。前面介绍了使用装饰器进行函数式复合的方法。在许多情况下，创建可调用的对象可便于在运行时将这些函数绑定在一起。

- **高阶函数**：Python 有许多使用其他函数的内置函数，包括 `map()`、`filter()`、`min()`、`max()`和 `sorted()`。此外，`functools` 库和 `itertools` 库中还包含其他一些示例。
- **Map-Reduce 算法**：很容易通过高阶函数构建得到。在 Python 中，它们相当于 `reduce(f, map(g, data))`的变体。可以使用函数 `f()`来处理归约，使用函数 `g()`来执行逐项映射。常见的归约例子包括 `sum()`，以及 `statistics` 库中的许多函数。
- **惰性（非严格）求值**：例如 Python 生成器表达式。像表达式(`f(a) for a in S`)就是惰性的，只会在客户端操作取值后才对 `f(a)`进行求值。许多示例使用了 `list()`函数从惰性生成器中取值。
- **单子**：由于在 Python 中对运算进行排序是不可避免的，因此通常不必强制指定执行顺序，可以使用 `pymonad` 库提供的一些显式语法来清楚地说明如何在更复杂的表达式中进行排序。这有助于输入和输出，对具有状态化行为的复杂模拟也是有益的。

除了这些常用的函数式编程设计模式之外，Python 中还有其他一些技术适用于函数式编程：

- **将尾递归转换为 `for` 语句**：Python 对递归设置了上限，且很少有循环允许超过此上限。更重要的是，递归涉及管理栈帧的开销，这在 `for` 语句中是可以避免的。
- **可迭代函数**：使用 `yield from` 语句可以轻松创建出由其他函数结果组成的可迭代集合函数。使用可迭代对象结果有助于函数式复合。
- Python 装饰器和可调用对象可以作为**函子**。在类 ML 语言中，函子用于将类型定义作为参数。在 Python 中，类型定义通常是基于类的，而且明智的做法是将这些定义与可调用对象或装饰器结合起来。

所有这些函数式设计模式都可以描述为设计和实现函数式编程的典型方法或常用方法。任何一种频繁重复的设计都会形成一种模式，我们可以从中学习并将其应用于自己的软件设计。

16.9　小结

本章介绍了三种优化技术。第一种技术涉及寻找正确的算法和数据结构，这对性能的影响大于其他单一设计或编程决策。使用正确的算法可以很容易地将运行时间从几分钟减少到几分之一秒。例如将使用不当的序列更改为合理使用的映射，可能会将运行时间缩减至 1/200。

通常应该将所有递归优化为循环。在 Python 中这样做会更快，并且不会因为 Python 的调用栈限制而终止。前面有许多将递归扁平化为循环的示例，尤其在第 6 章。此外，还可以通过其他两种方式保证性能。可以用记忆化来缓存结果。对于数值计算，这可能会产生很大的影响；而对于集合，影响可能较小。或者用可迭代对象替代大型实例化数据对象，也可能会通过减少所需的内存管理量而提高性能。

本章的案例研究展示了使用 Python 进行 EDA 的优势，即包含少量解析和过滤的初始数据获取。在某些情况下，需要大量的工作来归一化来自不同源的数据，这是 Python 的专长。

计算 χ^2 值涉及 3 个 `sum()` 函数：两个中间生成器表达式和一个最终创建期望值字典的生成器表达式。最终的 `sum()` 函数创建出了统计信息。在 10 多个表达式中，我们创建了一个复杂的数据分析用于协助接受或拒绝零假设。

本章还求解了一些复杂统计函数：不完全伽马函数 $\gamma(s, z)$ 和完全伽马函数 $\Gamma(k)$。不完全伽马函数包含一个无穷级数，我们将其截断并求和。完全伽马函数有一定的潜在复杂性，但并不适用于这种情形。

使用函数式方法可以编写出简洁明了的程序来完成大量运算。Python 不是纯粹的函数式编程语言，有时需要使用一些命令式编程技术。该限制迫使我们远离纯函数式递归。由于被迫需要将尾递归优化为显式的循环，因此可以获得一些性能上的优势。

此外还介绍了采用 Python 混合函数式编程的许多优点，特别是使用 Python 的高阶函数和生成器表达式提供了许多编写高性能程序的方法，并且这些方法非常简洁清晰。

版 权 声 明



微信连接



回复“Python”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

Python不仅拥有命令式编程的强大优化能力，而且还具备函数式编程的诸多特性。本书通过Python诠释函数式编程的核心思想，详细介绍如何利用函数式编程的优点，编写代码简洁明了且易于维护的高性能Python程序，充分释放Python潜力。各章由浅入深，循序渐进，全方位展示Python函数式编程的强大与精妙，助你迈向高阶Python开发。更有丰富代码示例，让你快速上手，学以致用。

- 函数式编程的基本概念与特性
- 用Python的内置函数操作数据集
- 使用递归设计算法以及常用归约函数
- 多个实用模块和PyMonad库
- 避开Python严格求值顺序的方法
- 常用优化技巧
- 使用迭代器和生成器表达式
- 常用高阶函数以及新建方法
- 元组处理技术
- 用装饰器构建复合函数
- Web服务设计方法

Packt>
图灵社区：iTuring.cn
热线：(010)51095183转600
分类建议 计算机/程序设计/Python
人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-52017-3
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks